Verifiable Computer Security and Hardware: Issues

William D. Young

Technical Report 70 September, 1991

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703
(512) 322-9951

This work was sponsored in part at Computational Logic, Inc. by the Naval Research Laboratory, contract number N00014-90-C-2351 and is the final report on Task 1 of this contract. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Naval Research Laboratory, or the U.S. Government.

Abstract

This report explores the influences of hardware on verifiable secure system design and envisions a mutually beneficial collaboration between the hardware verification and security communities. Hardware verification techniques offer the possibility of significantly enhanced assurance for secure systems at the lowest levels of system design and implementation. Security can provide an important and challenging applications arena in which hardware-oriented formal approaches can be tried and refined. We discuss some of the important concepts and issues that arise in trying to apply formal techniques to secure systems at the hardware level: the meaning of ‘‘security’’ in the context of hardware; the way to identify appropriate security properties at each level of system description, including the hardware level; and, a number of specific concerns related to hardware and its use in secure system development.

1. Introduction

The application of formal methods has been firmly entrenched as an approach to enhancing system security at least since the formulation of the A1 and beyond-A1 level requirements of the *Trusted Computer Systems Evaluation Criteria* (TCSEC) [99]. However, the formal analysis applied in secure system development to date has typically been carried out at rather high level of abstraction, leading some to question its usefulness and cost effectiveness in finding and eliminating security flaws in implemented systems. Some current work in verification [9] suggests that techniques and tools are becoming available for formal analysis at a level much closer to the actual implementation of a secure system. Formal analysis is addressing system issues that were previously intractable; distributed systems [28], real time systems [68], and particularly hardware [29, 79, 127] are coming within the scope of formal techniques. Future secure system development efforts could profitably apply these emerging technologies to further enhance system security.

Hardware is an obvious area in which to apply the latest formal analysis techniques. The *selection* of an adequate hardware base is obviously crucial in secure system development. It must be possible to build the protections mandated by the security model on top of the mechanisms provided by the machine. It is unlikely that the protection structures of the machine will match so closely the needs of the model as to be totally sufficient. But it is crucial that they not be incompatible with the requirements of the model. Consequently, attaining the highest possible assurance of security necessarily involves scrutinizing the hardware base upon which the secure system is built.

The *correct functioning* of the hardware is also crucial to the behavior of secure systems built on top of that hardware. Failures of hardware can seriously impact security. For example, one study [93] of the IBM 360/50 found ‘‘a total of 99 single-failure hazards [...] in the storage protection hardware and three [...] in the Program/Supervisor state logic. Any of these hazards could compromise the system security without causing a system crash’’ [130].

We envision a collaboration between the hardware verification and security communities that can be mutually beneficial. Hardware verification techniques offer the possibility of significantly enhanced assurance for secure systems at the lowest levels of system design and implementation. Security can provide an important and challenging applications arena in which hardware-oriented formal approaches can be tried and refined. Attaining assurance of correctness at the hardware level is an important but largely neglected area of concern in secure systems research. A number of studies have concentrated on the hardware security features of particular machines [26, 27, 48, 50, 62, 104, 118, 122, 129, 135]; several others [47, 49, 50, 64, 77, 78, 93] have focused on the general relationship between hardware and security. But almost no work has been done on the use of formal verification techniques to enhance the security of the hardware support for secure systems.

This report discusses some of the important concepts and issues that arise in trying to apply formal techniques to secure systems at the hardware level. In the following section we discuss the meaning of ‘‘security’’ in the context of hardware and the way in which we can identify appropriate security properties at each level of system description, including the hardware level. Section 3 describes a number of specific concerns related to hardware and its use in secure system development. The final section contains some conclusions about the use of formal approaches to enhancing system security at the hardware level of system design.

Many of the issues that we discuss in this paper deal with the use of general-purpose hardware used in the construction of secure systems. Various other issues arise when dealing with custom, application-specific hardware. For these, our discussions of security models, memory management, process support will not generally apply. The specific issues which do arise for these types are hardware are beyond the scope of this discussion.

2. Defining Security for Hardware

This study is principally concerned with the following two questions.

- How can hardware help or hinder system security?
- Can formal methods be usefully employed at the hardware level to enhance system security?

To address these, we must first consider two more basic questions.

- *What does it mean for hardware to be secure?*
- *Is it even meaningful to refer to* secure *hardware?*

These are the questions we address in this section.

Security is a property of an information system as a whole, rather than of particular components of such a system. Portions of the system may have specific properties whose maintenance is critical to system security; these portions may then be described as *security-relevant* or *security-critical*. However, it is the system as a whole that is either secure or un-secure. Consequently, it is somewhat misleading to discuss security properties of hardware, unless, of course, the hardware in question constitutes a complete information system. Still, it is commonplace to refer to the security properties of a system component and we will use that terminology unless there is any danger of confusion. Keep in mind that we are referring to those properties of the system components that are required to *support system security*, whether or not they would be naturally identified as ‘‘security properties’’ in any obvious sense. We discuss this issue further in the following subsection. In Section 2.2 we consider the question of how to identify properties of hardware supportive of security and how to construct a specification that captures these properties. Section 2.3 discusses the role of formal techniques in assuring the security of systems. Finally, in Section 2.4 we talk about hardware verification specifically and how it can be an effective adjunct to other techniques used in enhancing the security of systems. Specific features of hardware and how they relate to security are discussed in Section 3 in the second half of the paper.

2.1 Security Models

Security for an information system is typically defined in terms of a *security model*, which is both an abstraction of a class of information systems and a characterization of what ‘‘security’’ means for that class of systems. The model may be very abstract (eg., the machine model used in the definition of noninterference [52]) or be fairly concrete and include specific security control *mechanisms* (eg., the Bell and LaPadula model [7]). At base, a security model is simply a *specification* of an information system (or class of systems) and its security properties.

> What we are calling models here are no different than what are called specifications in the conventional software engineering process—and this is an important point: security models are nothing more than (usually formal) requirements statements and specifications for the security properties of a system design. As such, they are subject to the usual desiderata for specifications. [115]

Note that this is not a universally accepted view of models. There is considerable murkiness in the security literature about just what constitutes a security model, and endless confusion arising from the use of the overloaded word ‘‘model.’’ Some argue that ‘‘model’’ should be thought of in the sense of a scientific theory (as Newton’s laws are a model of the motion of bodies). Others construe models in the sense of a mathematical theory—a collection of a formal structures and axioms about their manipulation. Some of the attempts to clarify this issue merely exacerbate the confusion. Bell [6], for example, argues vehemently that the Bell and LaPadula model is an ‘‘abstraction’’ (but without once saying *of what* it is an abstraction). Some of the best books on the subject of secure systems [35, 47] give numerous properties and examples of models without ever defining the term ‘‘security model.’’ This confusion and ambiguity leads some [58] to question the usefulness of the entire notion. We believe that our view of a security model as merely a specific type of system specification that characterizes security for the system in question is the most useful approach from a verification standpoint.

2.1-A Types of Security Models

Most security models assume a class of active entities (*subjects*) and a class of passive entities (*objects*). Frequently, subjects and objects have associated *security levels* drawn from some ordered set. The intent of the model is to constrain the flow of information among the subjects in the system in some way determined by the ordering. For example, in models based on military security the levels form a lattice [34] or potentially a collection of lattices [44]. Intuitively, information may only flow ‘‘upward’’ in the lattice. The model defines security in such a way as to formally capture this intuition.

Access control models attempt to limit the flow of information by specifying the types of access (e.g. read, write, execute) each subject may exercise with respect to each object. In its most general form, the security model of the system is conceptually a matrix indexed by subjects and objects and specifying what types of access each subject may exercise to each object. [60, 76] For example, subjects may have read access to files that are at or below their level in the lattice. The intent is to limit information flow by imposing certain operational constraints on subjects.

The Bell and LaPadula model [7] is an access control model that incorporates a number of rules designed to restrict the flow of information. Among these rules are the *simple security property* and **-property*—often summarized, respectively, as follows: a subject may only read an object at a level at or below his own level (in the security lattice), and a subject may only write to an object at or above his own level. Variants of the Bell and LaPadula model have been widely used in secure system development [3, 46, 55, 85, 102, 120] despite the fact that the model does not preclude some types of intuitively un-secure behavior (covert channels).

Information flow models tend to be more abstract than access control models. Rather than define security in terms of the permissible accesses of subjects, they define security in terms of the absence of certain prohibited information flows, without regard for the particular mechanism through which these flows may occur. Thus, un-secure information flow is proscribed, even if no action of any subject violates the access control rules of the system. Several of the most widely studied security models are noninterference and non-deducibility.

Noninterference [52, 53] is an information flow model. Subject *a* is noninterfering with subject *b* if no action of *a* can have any effect on subsequent actions of *b*. Any specific noninterference policy specifies pairs of subjects that must be *noninterfering*. A noninterference policy tends to be stronger than an access control policy since it precludes information flowing through mechanisms that are not easily captured in the framework of subject-object access. Noninterference is limited to deterministic systems. It has been used as the security model for at least one large system development effort [16, 119].

Non-deducibility [128] is a strengthening of noninterference suited to non-deterministic systems. The key idea is the following. Assume that subject *a* is prohibited by the policy from passing information to subject *b*. The system is non-deducibility secure if any possible set of observations of the system by *b* is *consistent* with any possible set of actions of *a*. Daryl McCullough [87] has proposed a variant called *restrictiveness* that has the so-called ‘‘hook-up’’ property—two restrictive systems can be combined to yield a restrictive system. This model of security is used in the Ulysses (later called Romulus) system [86, 111] of Odyssey Research Associates.

A large number of other security models [21, 30, 34, 65, 88, 108, 131, 132, 134] have been proposed. Some have been general purpose and others designed for fairly specialized applications. Most have been concerned with preventing information disclosure; but others have been concerned with *integrity* of information in the system [13] or with preventing the denial of service to legitimate users [51]. In real secure system development efforts, most of these have given way to variants of the Bell and LaPadula model or to noninterference and restrictiveness. Some systems have used a combination of models; for example, the LOCK system [119] uses noninterference for the most abstract characterization of security for the system and uses an access control policy at lower levels of abstraction.[1]

[1] The reader will undoubtedly notice that there are many references in this paper to the LOCK system, developed by Honeywell and the Secure Computing Technology Corporation. We use LOCK heavily in our discussion both because it is a well-designed and innovative system and because it is the secure system development effort with which we have been most heavily involved and of which we have the most direct experience.

This discussion has concentrated on models for information systems as traditionally defined in ‘‘mainstream’’ computer security, i.e., mainly focused on the security of operating systems. There are many applications that are considered to be within the purview of security research[2] but that are not operating systems. These include secure distributed systems and networks, secure database systems, and various specialized devices such as crypto boxes and ‘‘guards.’’ A considerable amount of attention is being directed to extending existing models to cover networks [100] and to using the facilities provided by secure operating systems to construct secure database systems [39]. Specialized devices require specialized models that may be quite different than those discussed above as suitable for operating system security. Some very successful modeling and verification efforts [73, 121, 124] have been directed at such applications.

The choice of hardware support for a secure system will likely depend on the model of security to which the system must conform because the model delimits the legitimate actions of any subject/process.

> The choice of protection model is important because actions of programs operating on behalf of different users have different effects whenever different protection models are used to authorize those actions. For example, programs representing two different users may read and write a segment in a model that allows controlled sharing, such as the Access Matrix (AM) model. However, if those programs operate at different security levels, as defined by some Multilevel Security (MLS) model, then either one or the other of the programs is prevented from writing the segment. Thus, the action of writing a shared segment is a legitimate action within one model and a protection violation within another model. [48]

Certain types of hardware support are better suited to enforcing certain types of restrictions than others. Thus, in choosing a hardware base for a secure system, it matters what types of restrictions are intended. These in turn are determined by the model. In Section 3 we discuss specific hardware mechanisms and their relationship to particular types of security concerns.

2.1-B Properties of Security Models

As noted above, any specification of the system or class of systems defining the desired security properties of the system. As with any system specification it can be very formal or quite informal, abstract or concrete, complete or partial. The value and usefulness of the model depends upon each of these choices.

Ideally, a model should be formal. An informal model may be ambiguous, inconsistent or both. A formal model allows the application of rigorous reasoning techniques and tools to establish the consistency of the model and the conformance of the system model with its security requirements. [59]

Also, a model should be abstract. The careful use of abstraction can yield important clarity and intuitive structure to the system description and can provide significant help in formal analysis [67]. However, providing an appropriate level of abstraction can be a challenge. A model that is too concrete will often characterize security implicitly in terms of the mechanisms that are designed to provide it and thereby restrict the options of the implementor. A security model that is too abstract, on the other hand, may be too far removed from the realities of system design to provide much practical benefit.

The *style* of abstraction is important as well. We believe that it is dangerous to provide abstraction by neglecting system functionality in the model, i.e., by leaving the model incomplete. Completeness means that the model contains, in abstract form, all of the important essentials of the implementation. This issue is discussed further in the following subsection.

[2] Information security (INFOSEC) is sometimes defined to be the combination of computer security (COMPUSEC) and communication security (COMSEC).

2.2 Security Properties of System Components

Recall that the security model for a system refers to the secure operation of *the system as a whole*. How do we move from a system level model to appropriate specifications for the components that are comprised by the system, and ultimately to properties of the underlying hardware?

Assuming that we have defined an appropriate security model (specification) *for the system as a whole*, we can refine this into properties for the system's components in a very disciplined way. Since system behavior is determined by the behavior of the components and by the interactions among them, system properties, including security properties, must be implied by properties of the components and information about their interconnections. This suggests that the appropriate methodology for building secure systems is to begin with a high-level specification of the system and its desired security properties and to work downward toward an implementation. At the highest level of decomposition, the security properties of the system should be implied by (be provable from) the architecture of the system and the top-level properties of the component modules. These modules can, in turn, be refined into lower level modules whose properties should be those required to support the higher level abstraction. Thus, the properties of all modules in a secure system are *derived* from the requirement to support the properties of the level above. In other words, the security-relevant properties of any module in the system are exactly those properties that are required to support the properties of the architectural layer ''above.''

Of course, it is not usually practical in any real system development effort to follow a completely top-down approach. There is always a concern in any development effort that the lowest level hardware be adequate to support the requirements of the top-level specification. It is impractical to believe that the structure of the hardware can be completely ignored until all of the top level design issues have been thoroughly explored. However, we advocate as a general rule that our specification effort proceed top-down, with an informed understanding of what's below, and a willingness to iterate the process if the derived requirements are unimplementable on the available hardware base.

Deriving the specifications of system modules from the requirements of the calling environment is an obvious and intuitive approach to system development and one that has been institutionalized in, for example, the Gypsy Development Methodology [123].[3] Nevertheless, we are not aware of any actual sizable secure system development that has followed this approach. Used consistently in secure system development, it has the following important but possibly surprising consequences for defining the security properties of the various components of a secure system.

- This approach to secure system development will automatically delimit the ''security perimeter'' of the system.
- The highest level system specification should be ''complete,'' in the sense of covering all system functionality.
- It is usually a mistake to attempt to define the security properties of low level modules *a priori*.

Let's elaborate on these points further.

2.2-A Determining the Security Perimeter

A common approach to building a secure system is to partition the components of the system according to a perceived ''security perimeter'' and apply rigorous analysis only to those system components lying within this perimeter. This security perimeter is often referred to as the ''trusted computing base'' or TCB and ''serves to encapsulate all the security-relevant features of a system: nothing outside the TCB can impact the security of the system, and only the TCB has to be verified'' [81]. Failure of modules outside the TCB may adversely impact system performance but will not result in any breach of security. The process of determining which aspects of the system are security-relevant is typically an informal and *ad hoc* process; the perimeter may have to be continually readjusted as the security implications of various system components' behaviors becomes more clear [113]. An overly lax appraisal of which system components

[3] Some similar work formally deriving requirements from higher level specifications is reported in [94, 95].

are security-relevant may leave genuine security flaws in the system. An overly conservative drawing of the security perimeter will lead to costly and unnecessary analysis.

Our approach of using top down system specification and development leads to a formal derivation of the security perimeter. System modules within the perimeter are exactly those whose constraints contribute to the proof of the system-level security properties. Modules that can be left unconstrained when proving the top-level security property do not contribute to system security, and, conversely, cannot interfere with system security. Therefore, they can be safely excluded from formal analysis; they are truly outside the security perimeter. Modules that must be constrained in order for the higher level properties to be assured are those that contribute to the security of the system and that could possibly compromise it. They must be inside the security perimeter. By following a careful top-down style of security analysis, we automatically gain the very significant benefit that one of the most important and error-prone tasks of secure system development—determining the security perimeter—comes for free.

2.2-B Coverage of the Top-Level Model

Our approach to deciding what properties are required of the various components of the system assumes that the top level specification encompasses the *complete* secure system. If we specify only a portion of the system under some preconceived notion that some components are not security-relevant, it may turn out that our analysis will fail to uncover potential insecurities in the excluded portions. Thus, it is dangerous to leave out system components and functionality in the highest level description. Of course, we may still have intuitively un-secure behavior in the implementation due to a security model that is too weak to exclude it. But we will not have behavior counter to our security model because of security-relevant interactions with system components that were incorrectly believed to be security neutral.

Our highest level model must be ‘‘complete’’ for another reason. If we leave out aspects of the system in our highest level model, requirements on the system will likely force the addition of functionality in lower levels that is not reflected at all in the high levels. Thus, even if the high level design is provably secure, there may be additional capability in the implementation that can be exploited and that is accessible to a determined attacker.[4]

That said, we realize that it may initially seem unreasonable to reflect the entire system functionality in the highest level model. In doing so, don’t we risk losing all of the benefits of abstraction? There are various ways of introducing abstraction into a formal specification—structural, behavioral, data, and temporal abstraction. [90] In each of these, we can use the abstraction mechanism either for safely *hiding details* in the model or for unsafely *eliminating details* from the model. Various abstraction mechanisms tend to support one or the other of these approaches. For example, procedural abstraction in Ada (possibly implemented by interface assertions) allows suppressing the body of a routine while leaving the interface information for the routine complete and correct. This hides detail but without losing information that may be crucial to analyzing the correctness and security of a system.[5] For secure systems, we believe that any method of abstraction that loses information by selectively neglecting portions of the systems that may in fact be security-relevant is a dangerous luxury. This is because security is different than many other system properties that may be of interest.

Consider, for example, the correctness of a compiler. The purpose of a compiler is to *implement* the abstractions provided by a high-level language on the support provided by a lower level language or machine. The compiler is correct if each syntactically permissible program of the source language is translated into a syntactically correct and semantically equivalent program of the target language. However, there is nothing in the compiler that prohibits the machine user from accessing the lower level

---

[4]An example of how the use of a high-level specification (of a password protection scheme) can hide exploitable insecurities at a lower-level is described in [138].

[5]Of course, this is not quite true because of the possibility of non-local referencing and the probability that the system is accessible other than through the Ada compiler. Some languages, such as Gypsy [123] are designed in such a way that *all* of the potential effects of a routine are completely delimited by its interface specification. This supports the Gypsy *Independence Principle*: the specification of any Gypsy routine must be provable solely from its body and from the externally visible specifications of the routines it calls.

directly and performing actions that are not specifiable in the higher level (abstract) language. In fact, facilities are often added to high level languages to permit linking with assembly language routines. That is, the user's access to the machine is not limited to the abstractions provided by the high-level language through its compiler.[6]

Consider also the construction of safety-critical systems. Building secure systems is often compared to building safety-critical systems. Many of the same concerns for high assurance apply and many of hardware features that are potentially troublesome from the point of view of safety are of concern in security. For example, the British Ministry of Defence 00-55 draft standards [91] prohibit the use of certain types of hardware features, such as interrupts, in safety critical applications. However, this does not mean interrupts are not permitted on the hardware base on which a safety-critical system is constructed, but merely that they may not be used by the safety-critical system. Again, the machine user is not prohibited from accessing ‘‘dangerous’’ features; he simply may not access them via the abstractions provided by the safety critical system.

In contrast to these applications, security is not merely a collection of high level abstractions that are to be implemented by the lower level machine. Security is a property of the lowest level implementation, and it is crucial that the implementation perform the secure operations of the model *and nothing else*. If a hardware feature is dangerous from a security standpoint, not only may it not be used by the secure system, it should not be accessible on the hardware platform at all. A secure system must be designed with the presumption that if there is an available avenue of attack, malicious users will find and exploit it. It is this ‘‘and nothing else’’ constraint that makes it crucial that all of the possible actions of the implementation be reflected into the highest level models of the system to whatever degree possible. Otherwise, there may be accessible functionality at the low levels that can be exploited to violate security, but functionality which is not represented in the high level models. The fact that it is not represented in the high level models means that our high level analysis cannot find and exclude it. Our high level security specification may be *correctly implemented* and still intuitively un-secure.

This suggests, by the way, that ‘‘implements’’ as that term is often used is too weak a relationship between a security model and its realization. If implements is defined in terms of a commuting diagram such as that shown in Figure 1, this is inadequate to establish the security of the system.[7] The proof of such a commuting diagram shows that the abstractions defined by the high-level system are ‘‘implemented’’ by the lower level, but not that dangerous features of the lower level support are not accessible outside the abstractions defined by the high level. An adequate proof of security requires that the two level of abstraction be formally *isomorphic* in the sense that any insecurities in one are reflected in the other. Implements may be necessary for security but not sufficient.

Only methods of abstraction that manage complexity without discarding it are adequate for secure system development. We believe strongly that by careful use of procedural and data abstraction and by structuring a system design appropriately, it is possible to achieve both completeness *and* abstractness. Good [57] illustrates an approach for a simple secure system which meets these aims.

We should note that a quite successful approach to introducing abstraction into the specification of a secure system while preserving completeness was taken on the LOCK project [19, 137]. There, the highest level (abstract) models were used principally to establish and refine the security properties of the system. The crucial proof of the system security properties with respect to a formal description of the complete system interface was actually carried out at the Formal Top-Level Specification (FTLS) level against a formal characterization of security appropriate to that level of abstraction. This is the highest level of abstraction at which the full interface to the trusted computing base is visible. An informal mapping was given to higher level models, but the proof of security of the system did not depend upon the correctness of this mapping because the model was fully redefined at each level in the hierarchy. Therefore, the proof of security was carried out with respect to a relatively complete representation of the system (as we have insisted must be done) and a definition of security appropriate for that model. The LOCK approach preserves many of the

---

[6]There have been attempts to enforce security by permitting users to access a machine only through a high level language. [135] Such approaches are vulnerable to any means of directly accessing the machine at the assembly language level.

[7]This is the style of proofs done on the CLI stack [9].

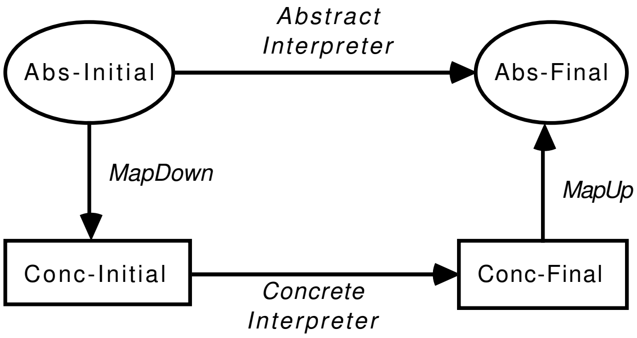

Figure 1: Implementation of Abstractions

benefits of abstraction as a tool for developing and honing intuitions about the secure system and its correctness properties, while gaining the benefits of ‘‘completeness’’ in the model.

2.2-C Modeling from the Top Down

Our final point noted above is that it is usually futile to attempt to specify a secure system from the ‘‘bottom-up.’’ This is because we cannot know what properties are required of a module at the bottom of the call tree until we know its role in the system. The security properties of a module may depend upon the security properties of its component modules, upon their functionality, or on both. Thus, the specification of a module is driven by the requirements of its calling environment.

Consider, for example, how to determine what requirements should be placed on the utility that retrieves a file in a multi-level file system. This is arguably a security sensitive function and so must have some security properties. However, we can imagine that the level and access control information relating to a file could be maintained in either of two places: physically with the file or in a separate table. Fetching a file (securely) from disk could be implemented in at least two different ways, depending on where this information is stored.

1. Assume that each file is stored along with its associated security information (possibly in the file header). We can fetch the requested file, check its associated security information against that of the requesting subject and deliver the file if the access permissions match in the appropriate way. Notice that no security violation occurs *even if we fetch the wrong file*.
2. Alternatively, assume that the security control information is stored in a separate table. We check the recorded level of the file against that of the requesting subject; if the subject has appropriate access rights, we fetch the file and deliver it to the subject. Here, it is vitally important that the file delivered is the same file as that whose level we checked in the table.

In these two cases, the primitive **`get_file`** utility will have different required properties in support of system security. In the first case, there is no need to guarantee that the file returned is the one requested (though we would certainly like that to be the case). In the second case, we do require that property. In both cases, the required properties are dictated by the security requirements of the calling environment.[8] Beginning at the ‘‘bottom’’ and specifying the routine without clearly understanding its role in the system and the way in which it contributes to the overall goal of secure system operation is likely to result in a specification that is not adequate to the goal of verifiable security.

Experience in building secure systems bears out this observation. On the Honeywell LOCK project, for example, it was found that the required properties of the security co-processor (SIDEARM) were ‘‘security’’ properties and those of the kernel interface software (KIS) were largely ‘‘functionality’’ properties.

---

[8] A similar point is made by Neely and Freeman [101] who call modules that contribute to the security of a system ‘‘trust domains’’ and note that the constraints on each trust domain are determined by its environment.

> The verification of the KIS should be somewhat simpler than that of the SIDEARM. The verification is functionally-oriented as opposed to property-oriented, as in the case of the SIDEARM. One only needs to verify that the KIS adheres to some low-level properties. For example, the KIS must be shown to totally clear the CPU registers during a context switch. [119]

Of course, it is largely a matter of opinion whether any specific property should best be regarded as a security property or a functionality property. Clearing of CPU registers could be regarded as a security property since it enhances the security of the system. Our point is that the specification of the module is more likely to be ‘‘the registers have been cleared’’ rather than ‘‘the KIS is secure’’—in the words of the quote above, ‘‘functionally-oriented’’ rather than ‘‘property-oriented.’’

We believe it likely that the smaller the module and the closer it is to ‘‘raw’’ hardware, the less likely it is to have identifiable security properties and more likely to have required correctness or functionality properties. At the very lowest level of abstraction, it is difficult even to imagine what a security property looks like; the notion of a secure *AND* gate is hard to envision. For this reason, any attempt to specify security-specific properties of low level hardware modules is likely to fail. This suggests an answer to Don Good’s question of what a *secure wire* looks like [58]; it probably looks exactly like a *correct wire*.

This is not to say that we cannot identify security-relevant properties or hardware, only that the most security relevant property of hardware is often that it behaves in a functionally correct fashion. In support of this position we quote the findings of a recent report by the National Resource Council on hardware influences on computer security: ‘‘the only essential, then, is to have simple hardware that is trustworthy’’ [97].

Of course, some hardware features have been correctly identified as generally useful in the construction of secure computing systems. We will address a number of such features in Section 3. We merely caution that defining what it means to be a secure system, a secure module, and particularly secure hardware requires more care than is often given. A ‘‘secure adder’’ or a ‘‘secure UART’’ is largely meaningless except in the context of a larger system.

2.3 Formal Verification and Security

Much of the experience in applying formal techniques in secure system development to date has been driven by the requirements of the National Security Agency’s *Trusted Computer Systems Evaluation Criteria* [99] (Orange Book). It is unclear how much of this experience is applicable to verifying secure hardware.

The approach mandated by the Orange Book requirements takes the following pattern:

- Formulate an abstract model of security for the system and analyze it using currently available formal techniques.
- Devise a Formal Top Level Specification (FTLS) for the system and show by ‘‘a combination of formal and informal techniques...that the FTLS is consistent with the model.’’
- Show correspondence of the FTLS to the TCB source code by ‘‘manual or other mapping.’’
- Conduct a covert channel analysis involving the use of formal methods.

This pattern of specification, illustrated in Figure 2, has become institutionalized by the promulgation of the A1 level requirements and is sometimes taken as illustrative of state-of-the-art formal methods in the security arena [8]. We continue to expect and accept this style and level of analysis for two major reasons.

- The common wisdom holds that this is the best we can do; code level verification of large systems is deemed beyond the current state of the verifier’s art. The feeble mechanical support provided by the tools currently appearing on the NCSC’s Endorsed Tools List [98] fosters this impression.
- The driving force behind contractor adoption of formal techniques has been the list of certification requirements in the Orange Book. By institutionalizing the *status quo* we have eliminated much of the motivation for innovative approaches to enhanced assurance.

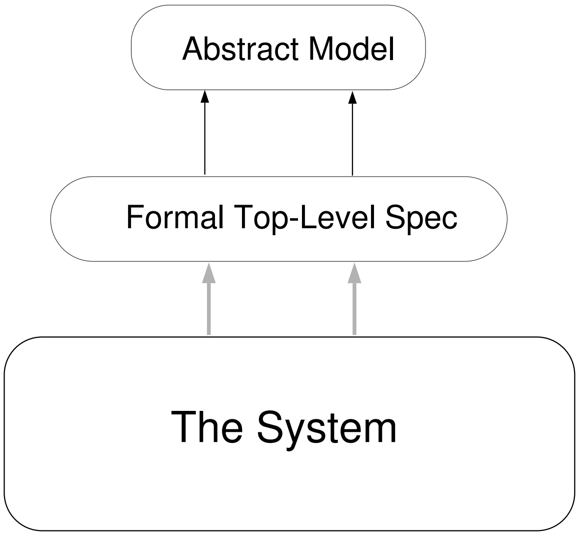

Figure 2: Current Secure System Verification

There are good reasons for questioning the usefulness of this style of high level analysis. Rather than reasoning about implemented systems, it largely involves reasoning about the *specifications* of systems. We hope that these specifications will be reflected in the implementation in such a way that insights gleaned from the formal analysis will affect the ultimate quality of the system. Yet the link from the formalism to the nuts and bolts system is often a rather tenuous correspondence argument, the believability of which relies on an understanding of the semantics of *two* abstractions—the specification language and implementation—either of which may not be fully formalized. Even in those situations where the formal analysis has a significant influence on the actual design, the problem of establishing a convincing correspondence between a high-level Gypsy specification and a low-level C implementation, say, is daunting [138].

It has even been suggested that design-level analysis may actually be dangerous by fostering unwarranted assurance in ‘‘verified’’ systems. This perception is exacerbated by the fact that the formal methods community has been too glib in the past about stating what has been and can be accomplished with formal methods. Enthusiastic pronouncements by advocates of formal methods have been taken out of context [45] to provide a misleading picture of the goals and expectations of formal analysis. Marketing claims made regarding some ‘‘verified’’ products may not have been justified by the actual formal analysis. [24, 31] This again has the potential for fostering unrealistic and ultimately deflated expectations. Whereas we believe that design level verification has a role in improving the quality of systems, we feel that the technology is now available to combine these high level approaches with much lower levels of analysis. Particularly for safety-critical and secure systems, hardware verification can play a role in delivering very high assurance systems.

2.4 The Role of Hardware Verification

Hardware verification is a rapidly expanding research area. Researchers have considered verification of hardware at various levels of abstraction from circuit designs written in high-level hardware description languages such as VHDL [84, 117] down to low-level models of MOS circuits [25, 133]. Techniques have been developed that are adequate for handling small circuits [5] to handling entire microprocessors [66, 69]. There is a rapidly growing literature in this area [29, 79, 127, 136].

In many ways, verifying hardware is a more tractable problem than verifying software because hardware tends to be more regular. However many of the concerns are different and others are potentially much worse in hardware. For example, pointers are traditionally quite difficult to reason formally about and many ‘‘verifiable’’ languages such as Gypsy exclude pointer manipulations. However, addresses are merely pointers and are pervasive in hardware operations. Consequently, imposing a discipline on address manipulation is essential to formal treatment of hardware. Such a discipline is exactly what is provided by memory protection hardware. A simple base and bounds scheme can guarantee that a program’s access to memory will not extend beyond its permitted boundaries. Thus, the hardware mechanisms provide an essential structuring that can be relied upon by the verification of the higher-level protection mechanisms that are built on top of it.

This suggests the use of hardware verification techniques as a natural evolution of the application of formal methods in secure system development described in Section 2.3. The Orange Book A1 requirements call for ‘‘verification evidence consistent with that provided within the state-of-the-art.’’ The discussion of ‘‘Beyond (Class) A1’’ systems acknowledges that ‘‘consideration will be given to the correctness of the tools used in TCB development (e.g., compilers, assemblers, loaders) and to the correct functioning of the hardware/firmware on which the TCB will run’’ [99]. Hardware has not been entirely neglected in secure systems specification and verification. Both the SCOMP [46] and LOCK [16, 119] specifications included protection mechanisms that were implemented in hardware as well as in software. However, these efforts considered features largely modeled at a level of abstraction where the distinction between hardware and software does not matter.

The paradigm of secure system development illustrated in Figure 2 may be extended ‘‘downward’’ toward the hardware level as shown in Figure 3. There are significant research issues involved in relating levels of abstraction involving both verified hardware and verified software. The CLI ‘‘stack’’ effort [9] may offer some directions. However, as we indicated in Section 2.2, the correctness properties of a compiler or assembler are different than the security of a system; *implements* is simply the wrong relationship between the layers of the hierarchy in a secure system development. We are aware of only a few efforts [12, 14], other than the CLI stack, in which properties defined formally at a fairly high level of abstraction have been ‘‘pushed down’’ to the hardware level; none of these were explicitly security properties.

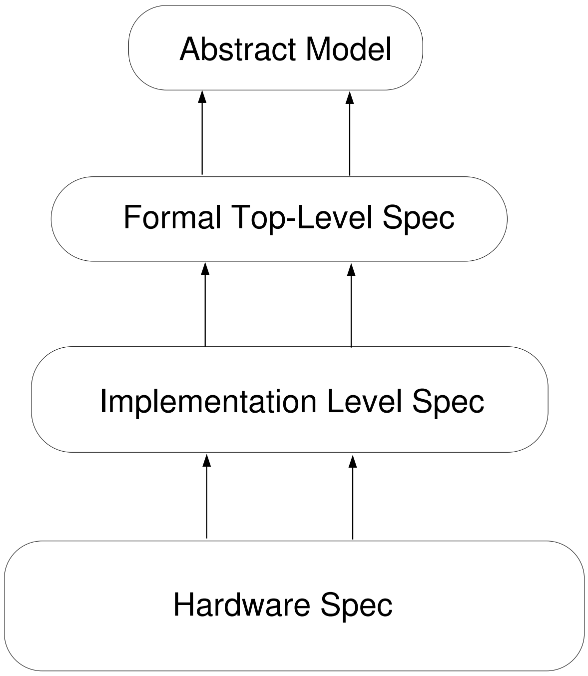

Figure 3: Proposed Secure System Verification

As an example of a significant and challenging research area where hardware modeling and verification could contribute consider hardware features such as memory protection mechanisms and support for process isolation. Such features often form a base upon which secure systems are constructed. For example, many of the security models discussed in Section 2.1 above take as primitive the notion of identifiable and separate ‘‘subjects’’ that are the active agents in the system. In implementations, these are often identified with processes supplied by the underlying operating system or hardware mechanisms. Process isolation, then, is crucially relied upon by the security mechanisms and taken more or less for granted in the model. Yet, such underlying mechanisms have often been found to be the source of security flaws in implemented systems [1, 2, 62, 83, 135].[9] Few serious attempts have been made to formally model and verify these crucial underlying facilities;[10] these facilities are obvious candidates for formal scrutiny at the hardware level.

But how can we go about assuring the correctness of such basic mechanisms? There is no fundamental difference between, for example, proving the correctness of a process isolation scheme and the correctness of some application built relying upon process isolation. Both require a clear and unambiguous (formal) specification and a proof that an implementation meets this specification. Some proofs have been done on specific instances of this problem; Bevier [11], for example, gives a specification for process isolation on a particular target machine. However, what is really needed is a *generic* axiomatization of memory and memory management, of process isolation, and of the other basic mechanisms of hardware support for secure systems in particular and operating systems in general, rather than treatments that focus upon specific security models or processor architectures.[11] Such notions as memory management are the underlying abstractions upon which security mechanisms rely and should be formally viewed as such. [67]

We are not aware of a rigorous generic axiomatization of features such as memory protection, though Popek and Goldberg [109] sketched the outlines of such a project in 1974. Such a presentation, submitted to the social process, would provide a convincing specification as a basis for formal analysis and a viable target for implementations. Landwehr and Carroll [77] provide a start toward a security-specific version of such a specification by enumerating logical requirements on an abstract machine. Their presentation falls short of a formal treatment of this issue but is a step in the right direction. It is rather surprising that more effort has not been expended in this area, given the importance of these issues for system correctness and system security. Such a formal treatment is outside the range of the current report, but is a potentially productive area for future research. This is a challenging research area and one calling for increased collaboration between the hardware verification and security communities.

3. Aspects of Hardware Support for Secure Systems

Various fairly standard features of microprocessors have traditionally been regarded as supportive of system security; these include memory protection schemes, support for interprocess communication, and support for multiple execution domains.

> It is fortunate that most of the hardware mechanisms needed to implement a secure system are also required by conventional operating systems; otherwise, there would be little hope of seeing these features on today’s machines. This is not a coincidence: good protection features are essential to an efficient and reliable operating system. [47]

This section describes a number of such features, the way in which they support system security, and the implications for verified secure systems. We make no claim that our list is complete in any sense. Almost any hardware feature accessible to an end user could have potential use in some covert activity. This motivated our insistence in Section 2.2 that our formal system model be as inclusive as possible. We focus

---

[9] Rushby [114] is the source of this list.

[10] Bevier [10] is the only attempt of which we are aware to model and prove the correctness of a process isolation scheme down to the machine code level.

[11] See [70] for some work on the generic specification of hardware.

on aspects of hardware that seem particularly relevant from a security perspective. There is no special order to the list of subtopics in this section. In the Appendix we give a prioritized list of hardware features which should be examined in a secure system development effort.

Notice that most of the features described in this section could be implemented in hardware, in firmware, or in software. Our choice to describe them as ‘‘hardware support for secure systems’’ is driven by the fact that they are usually implemented in hardware, chiefly to enhance system performance and to yield a cleaner and better structured architecture. There is no reason to believe that modeling techniques would be significantly different for a hardware implementation than for a software implementation.

3.1 Influence of System Architecture

One key factor influencing the verifiability of any piece of complex hardware is the structure of the architecture. It is well known from software verification experience that a well-structured program is more likely to be comprehensible and easy to verify than a poorly-structured program. The lessons of modularity and structured design arose in the same environments as formal program analysis techniques. These same lessons apply to hardware. Modular design and good engineering practice lead to hardware designs that are accessible to formal techniques.

An architecture that matches in structure the systems that are built upon it can greatly enhance their verifiability. Dijkstra [38] illustrated the benefits of a layered architecture for operating system design. Similar benefits accrue for a hardware architecture. For example, the hardware of the PDP 11/45 was layered to support three levels: kernel, supervisor, and user. A protection system that matches the hardware layering closely gains in comprehensibility and verifiability because the mechanisms in the protection system match closely those of the machine. Many researchers have recognized the benefits of localizing the protection functionality of the machine into a small *kernel*, the lowest level of the system layering. Popek and Kline [110] assert that the kernel should ‘‘contain the security relevant portions of the operating system primitive, and nothing else.’’ Spier [126] wants the kernel to contain only the interrupt and I/O handling routines, CPU multiplexor and scheduler, and memory management routines. ‘‘Kernelizing’’ the protection mechanisms isolates them from the influences of other hardware and software. In Section 3.3-E we discuss the notion of a reference monitor, one instantiation of kernelized protection.

One very significant benefit that can be gained by careful structuring of a system architecture is that the amount of verification required to gain a particular level of assurance can be reduced. High levels of assurance come at a price; formal verification techniques tend to be expensive and difficult to apply. Consequently, there is a strong economic incentive to reduce the percentage of system functionality that must be verified. This is much of the reason for the process discussed in Section 2.2 of delimiting the security perimeter, i.e., the goal is to exclude as much as possible of the system from formal analysis. The architecture of a secure system strongly influences how much of the system needs to be verified. For example, as we will discuss in Section 3.5 smart I/O device controllers can be a security risk and must be either trusted or verified on conventional secure system designs. This need was avoided on LOCK [119] by segregating nonprimary memory and communications ports from the CPU and main memory by a bulk encryption device. The result is that:

- contents of secondary memory are encrypted and need not be physically secured; and,
- device controllers need not be trusted, allowing the use of commodity controllers.

The result is a significant reduction in verification effort with no loss in system assurance.

Other architectures which use physical arrangement of system components to gain security and reduce the need for verification are discussed below in Section 3.3-E(2).

3.2 Memory Protection

One of the most crucial aspects of hardware support for secure system development is the ability to define distinct memory spaces for separate users/processes. Access to memory is usually restricted on modern processors by some scheme of implementing *virtual memory* [56, 109]. There are myriad variants of virtual memory management; but all have one or more of the following goals:

1. they isolate process address spaces;
2. they permit a process to have a larger address space than available main store;
3. they free the user from having to deal with storage management; and,
4. they utilize main store efficiently by maintaining unused portions of program and data on secondary storage until needed.

Virtual memory management schemes are aimed at high efficiency of resource utilization and satisfactory performance. The economies of computing have changed over the years as hardware (and particularly memory) costs have fallen, but these requirements remain.

Typical memory management schemes involve mapping user-supplied (virtual) addresses to physical addresses via some virtual memory mapping. This process is typically supported by hardware using index registers and address registers. This translation is transparent to the programmer. See any book on machine architecture (such as [63]) for further discussion of the details of memory management schemes.

Some early machines implementing virtual memory required that a process' address space lie in contiguous physical memory; physical addresses are computed from virtual addresses by adding an offset. This is simple to implement with base and bounds registers storing the limits of the accessible region. This scheme requires that all accesses check these limits and that the registers not be accessible to the user program.

Requiring contiguous memory leads to storage *fragmentation*. To avoid this, we can divide memory into fixed-size pages and allocate to each process either a fixed or variable number of pages. In *demand paging* schemes, pages are stored on secondary storage until required. This permits a process to have a virtual address space larger than the main store. To be efficient demand paging requires hardware support in the form of hardware mapping registers and page descriptors and efficient support for swapping pages in from secondary store.

Some systems add an alternative or an additional layer of memory division in the form of *segments*. The segments are typically variable sized memory partitions that may hold distinct objects or may be merely an extra layer of memory partitioning. A translation mechanism for virtual memory that accommodates variable-size segments on top of demand paging requires an additional level of memory descriptors. There will usually be hardware support for this mapping.

Virtual memory management is an obvious mechanism for controlling the access of users to memory objects. A crude scheme is to divide memory into user and system segments that are separately protected. The user may not write into the system segment. Functionality of the operating system is accessible only via special entry points. If this partitioning is static, this scheme is easy to enforce with hardware.

As we suggested in Section 2.4 above, virtual memory mechanisms and the hardware support for them are obvious candidates for formal scrutiny. They provide foundation for much secure system development effort but have been largely neglected in formal efforts to assure the correct functioning of such systems.

3.3 Execution Domains

Since security implies the isolation of information or other resources from unauthorized access, several authors such as Landwehr and Carroll [77] take as the key abstraction within a secure system the notion of a protection *domain*—''a set of information and authorizations for the manipulation of that information within a computer system.'' A domain is essentially the name/value space accessible to a particular user. With respect to security models, a domain may be identified with a *subject* and its associated access rights to objects in the systems. Important factors to consider in implementation of protection domains are:

- how the protection structures of the model map onto those provided by the machine architecture;
- how domains are initialized;
- how users are linked to domains;
- how communication occurs between domains; and,
- how domains are isolated from one another.

The principal concern of secure system developers is providing required domain isolation while still providing appropriate communication between domains and efficient provision of the other services of the machine such as creation and initialization.

3.3-A Processes

Within a given implementation, a protection domain may be identified with a process, with an abstract data type, or with an object (in an object oriented system). At the hardware level, it is often a *process* that is associated with a protected domain. This is natural because many conventional architectures provide fairly extensive hardware support for processes. Consequently, the hardware process support is often relied upon by the operating system as a foundation for the security mechanisms.

> Almost all security models take the notion of ‘‘per-process’’ virtual environments as primitive—so primitive, indeed, that this notion is generally unstated and unregarded. The key assumption is that the active entities of a security model (‘‘subjects’’ in security jargon) are assumed to be fundamentally distinct and distinguishable from each other. Now, in a concrete system design or implementation, the subjects of the model are generally associated with ‘‘processes’’ provided by the operating system nucleus and in order for the security model to be an accurate description of the concrete system, it is therefore necessary to ensure that the processes of the concrete system are indeed distinct and distinguishable—that is to say, each process must have its own ‘‘virtual’’ environment. [114]

Thus, the machine facilities that support the per-process virtual environments and keep them separate are often crucial to the maintenance of system security.

The main required support for processes is the ability to switch from one process to another. This requires a mechanism for storing enough process state so that a suspended process can be re-dispatched. For a multi-level secure system, if processes are associated with subjects, there are likely to be numerous processes and the need for frequent process switching. This calls for an efficient interprocess switching mechanism. This mechanism must store the process internal state including at least the program counter and user-visible registers. In addition, there must be a mechanism for preserving the user-addressable memory. The mechanism for this depends upon the particular memory management strategy of the machine.

3.3-B Initialization

Landwehr and Carroll point out the need for reliable system initialization in secure system operation.

> The ability to define and separate domains in a machine will be of little use unless there is a reliable way of getting started. This implies that the logical structure of the system must allow the programmer to distinguish the occurrence of a system initialization event and to establish a consistent state for the system—a state from which additional domains can be initiated and separated. [77]

From a modeling standpoint, initialization is crucial. Security properties are often stated as *invariants* on the system state; the system remains secure if each transition in the system is security preserving.[12] However, the implicit induction here requires that the system be initially secure. Also, because of the need

---

[12] See [89] for discussion on some of the subtleties of this approach to security.

at start-up to establish all of the security mechanism including the tables that contain access control information, the initialization mechanism must have privileged functionality that likely is not under the mediation of the security control mechanisms.[13] Thus, special care is needed in the initialization of the system to preserve security.

Hardware can contribute to secure system initialization in a variety of ways. Secure initialization routines can be isolated from other portions of the memory by using virtual memory techniques as discussed in the previous section, by storing security tables in ROM, or having by a processor dedicated to initialization. Via these techniques it is possible to ensure that a secure initial state is established. LOCK [119], for example, avoids the problem of having to bypass the security controls during initialization by having the entire security kernel segregated on a separate processor with its own memory.

Initialization can be a particularly difficult challenge when reasoning formally about hardware. For example, a recent proof about a hardware reset mechanism [72] found considerable difficulty in proving that a microprocessor reaches a ‘‘safe’’ and predictable state following a reset. Such issues are often neglected in formal approaches to hardware correctness.

3.3-C Linking Users with Domains

It is always necessary to associate the actions of an active entity (subject/process/domain) in a secure computing system with an individual user. Thus, it is necessary to have a reliable way to link users with domains, both when the domains are created and as they execute. Such accountability provides a strong deterrent as well as provides a way to diagnose security breaches *post hoc* through audit procedures.

Most systems implement a password mechanism to prevent unauthorized persons from gaining access. A password mechanism can also prevent users who are already logged in from performing unauthorized activities, such as accessing restricted directories and gaining access to more privileged ‘‘modes’’ such as super-user mode on Unix.[14] The user must authenticate the machine as well, to avoid so-called ‘‘spoofing’’ attacks in which a program masquerades as the login procedure on the target machine until a user divulges his password. Two way authentication can occur via a series of questions and answers. Hardware mechanisms that can assist in this process include devices in which the user uses some physical device such as a magnetic-striped key card rather than a typed identification. Such a login scheme for mutual identification of user and system is described in [43].

After access to the machine is obtained, the integrity of the connection must be maintained. This can be accomplished by periodically reauthenticating the user to ensure that the user has not changed and logging out users who have been inactive for some period of time. A hardware watchdog timer can be useful in implementing such schemes.

Related to the issue of linking users with domains is the question of what *audit facilities* are provided by the machine. Audit facilities are intended to record occurrences of selected system activities so that actual or attempted security violations can be attributed to particular users. An audit facility has the goals of allowing reconstruction of events leading up to and including system violations and allowing monitoring of user actions so that attempts to violate system security may be perceived and acted upon before a violation occurs. [92, 105]

There are two main issues related to audit data: data must be adequate to meet the audit goals of the system, and both the audit mechanism and storage of audit data must be tamperproof. Selection of auditable system activities is an art. Too much audit data can swamp the system and make extracting relevant information

---

[13] This is an instance of a more general problem — most security models are *too strong*. To operate well, most systems require functionality that is formally in violation of the security policy. For example, it may occasionally be necessary to downgrade documents, in violation of the proscription of downward flow of information (*-property). A general utility such as a printer demon may need to read and write documents at all levels, in violation of most security models. The solution is typically to have a collection of ‘‘trusted processes’’ which are privileged to violate the security policy in specific ways.

[14] See [96] for an interesting history of the password security mechanisms on Unix.

very difficult; too little audit data may allow a clever perpetrator to escape detection. Selection of auditable events is a system design rather than a hardware issue. However, hardware can be used to make recording audit data less obtrusive. In a system with hardware support for kernel calls, the hardware can also write the relevant audit data. This is generally much more efficient than software monitoring of system behavior. Moreover, it may add additional assurance that the mechanism is tamper resistant. Previously recorded audit data can be protected by the standard protection mechanisms of the system. However, for purposes of reconstructing *past* security violations, there is no reason to assume that audit data has not be compromised along with other system resources, if all were protected by the same mechanisms. Some system such as LOCK [119] store audit data on a write-once medium. This provides additional assurance of tamper resistance.

3.3-D Communication Between Domains

Communication between processes/domains is critical to security, since the entire purpose of a secure system is to control communication between subjects. Again, the support for and protection of communication between domains may not match the required level of protection called for in the security model. For example, if a hardware supported domain cuts across logical subject boundaries or if there is more than one level subject within the domain, communication must occur in a manner consistent with the policy.

Information can pass from domain to domain in a number of ways:

- one domain may request service from another and transfer control to it;
- two domains may share memory;
- domains may pass messages.

There are various ways in which hardware can support the restriction of interdomain requests. Spier [125] outlines a protection scheme that depends on ‘‘the availability of dedicated supporting protective hardware...where an inter-domain call is just as efficient as a common intra-domain call.’’ Required is a hardware return stack inaccessible to the user program and paging tables. On LOCK all permissible domain transitions for any subject are encoded in the Domain Transition Table (DTT). Hardware mechanisms support the checking of this table whenever an attempt is made to move from one domain to another. In some ring architectures, a domain change can only occur by transferring control through a **`call`** instruction to prescribed locations in special *gate segments* that are designated as entry points to inner rings [47]. Such mechanisms allow a protected domain to control its invocation.

Entering a less privileged domain requires that the arguments passed be checked to prohibit the calling process from illegally obtaining access to restricted data. Most such checks are performed by software. Hardware can help in address validation to ensure that passed pointers refer to legitimate locations within the user space. See [47] for an extended description of the various types of pointer validation and hardware supported schemes for carrying them out.

Other systems implement interprocess communication via shared memory or message passing. These approaches to communication are typically controlled by the assignment of *access rights* restricting what privileges any particular domain has to shared memory or to which other domains a message may be sent. These topics are covered in the following subsection.

3.3-E Isolation of Domains

Undoubtedly the most crucial function of a secure system is to enforce (partial) isolation of domains. Of course, a domain cannot be entirely isolated or it could never interact with its environment or communicate its results to the external world. However, the communication between domain must be constrained in such a way that only information flow allowed by the security policy is allowed in the system. This means that all other information flows must be prohibited.

Rushby and Randell [116] note that there are conceptually four ways to provide domain/process separation: temporally, physically, cryptographically, or logically. These methods of domain separation can be realized physically in a number of ways.

3.3-E(1) Temporal Isolation of Domains.

In the absence of any more sophisticated separation mechanisms, early machines utilized the *temporal* isolation of domains. That is, any user would have access to the complete memory space of the machine, but separate users were isolated in time, since only one user has access to the machine at a given time. Such machines either had no memory management software (eg. the IBM 1130 and Elliott 905) or a simple resident executive to control sequencing (eg. the IBM 7090, and scientific configurations of the Elliott 4130 and ICL 1900). On these machines, there is no sharing of resources among users. The special hardware required to implement this is only a fixed protected area of memory for the resident. User programs did not have access to this protected region to prevent corruption of the executive. This protection scheme is particularly easy to implement and to prove correct.

Another style of temporal isolation is enforced in some multi-user settings where the machine is utilized in ‘‘single-level mode’’ by groups at different security levels. After use by a high-level group, the store is ‘‘scrubbed’’ before the machine is turned over to a group at a different level. Security concerns arise when information on the system survives between uses. User-accessible store that has not been adequately cleansed may contain residual information that is accessible to the next user. Even if user-accessible store is cleansed, boot-strap routines, initialization software, and other system software provides a potential mechanism for preserving sensitive information from one use of the machine to the next, if the machine does not provide adequate protection of system space. Also, it is known that information can be gleaned from some physical media even after they have been logically sanitized. For example, information stored on a magnetic media can often be recovered even if it has been overwritten. Thus, tapes used for secure applications and subsequently retired must be ‘‘de-gaussed’’ to prevent retrieval of the stored information.

3.3-E(2) Physical Isolation.

An obvious approach to maintaining isolation between processes is to schedule them on physically isolated processors with minimal and carefully controlled opportunities for communication. This approach of using distributed computing as a means of attaining security has been suggested by several authors [32, 40, 112, 116]. The underlying motivation is that ‘‘manifest physical separation is one of the simplest and most easily verifiable forms of separation’’ [40]. The advantage of such schemes is that all interprocess communication mechanisms are simple and explicit and there is little need to fear covert channels through the system state. However, the efficiency advantages of shared state are sacrificed.

Physical isolation is used rather heavily in cryptological equipment. Red/black separation is realized in many COMSEC applications by isolating red and black system components on separate processors with a tightly controlled bypass. The verification of such a system is described in [124].

Another use of physical isolation is to scatter resources to make an attacker’s penetration of a single system component less profitable. Such a scheme is used by the SATURNE project [37]. File fragments are distributed to various storage devices to impede security penetrators.

A rather radical architectural approach using physical isolation to gain security is suggested by Davida, DeMillo, and Lipton [32]. They propose three different system architectures for which it is claimed that ‘‘the security of the total system derives from a few very simple hardware devices by socially acceptable arguments’’ (i.e., requiring no verification, other than the ‘‘social process’’ touted by the authors in another paper [33]). The basic scheme in its simplest incarnation is to segregate users at different levels on separate processors and to allow communication among processors via hardware-enforced one-way links. This structure physically enforces the information flow constraints of the model. The practicality of the scheme has apparently never been tested in practice.

With the advent of cheap yet powerful personal computers and workstations, there is an increasing tendency to return to the paradigm of single-user machines to enforce domain separation. This is a form of physical isolation. Like early machines, this obviates ‘‘internal’’ security mechanisms to isolate users from one another. However, networking such machines introduces many of the security concerns of multi-user computing on a single machine.

3.3-E(3) Cryptographic Isolation.

An adjunct to other approaches to security is to encrypt sensitive data. Encryption of secure data has the effect of reducing its effective information content. However, the effectiveness of encryption as a protection scheme is directly related to the ‘‘strength’’ of the encryption scheme and to the security of the key. Details of various encryption schemes are well described elsewhere [35].

Encryption is not a complete solution since the key data must be protected and potentially distributed. However, encryption can be useful in protecting data in remote storage or in transit. Remote access systems are particularly vulnerable to penetration. Not only is the computer susceptible to attack; transmissions between the computer and the remote user can be intercepted or subverted. End-to-end encryption, as is used in modern secure telephone technology, can give protection in remote access systems.

LOCK is an example of a secure system which uses cryptographic techniques in a variety of ways. As we saw in Section 3.1, LOCK uses bulk encryption to store secure data during periods of time when no program needs to access it. This reduces the need for physical security on the storage medium. ‘‘In addition, cryptography is used to close covert channels[15], protect security-critical data bases, and defend against attacks by subverted device controller hardware and firmware.’’ [17]

The use of cryptographic isolation may eliminate some of the verification requirements on I/O controllers and other portions of the system. However, it puts a correctness premium on the portions of the system that perform encryption and decryption. These units are often critical to system security. Yet the crypto devices are often of classified design and may be difficult to incorporate into secure systems.

> Free-standing cryptographic units have been difficult to integrate, both physically and functionally, into modern computerized hosts. The classified nature of the products has often imposed physical security constraints which are incompatible with operational needs. The ‘communications-only’ bias of the products has inhibited cryptographic solutions to computer security problems, such as authentication of critical but forgeable user/computer dialogues and the security of classified information on removable media. [74]

The use of non-classified modules designed for integration into secure systems may alleviate some of these problems. It also opens an additional applications area for hardware verification techniques. Some formal methods work has been done in the analysis of cryptographic techniques [23], but we are aware of no (open-literature) efforts on the verification of hardware *implementations* of these algorithms.

3.3-E(4) Logical Isolation.

Probably the most widespread approach to implementing domain isolation in modern secure systems is some form of logical isolation. The basic approach here is to assign security attributes to various processes/domains and to implement mechanisms to ensure that they can communicate only in very restricted ways. The logical separations and the allowable communication patterns must match those of the security models as described in Section 2.1.

3.3-E(4)-a Isolation via Access Permissions.

Many of the mechanisms discussed above, such as those that implement virtual memory (Section 3.2), are not specifically protection mechanisms, though they may be used to advantage by secure systems. The mechanisms in this section, segregating domains by limiting the *types* of access to objects that domains may exercise in the system, are specifically protection-oriented. They are specifically designed for protection, either from malicious or careless access.

---

[15]This covert channel illustrates the subtlety of formal secure system design. On LOCK, unique identifiers are generated from a counter as objects are created. If these uid’s were left unencrypted, a low-level observer might infer how many objects had been created in the *whole* system within some interval by observing the difference between the uid’s of objects he created at the beginning and end of the interval. This could permit a high level observer to signal down by modulating the creation of objects at a high-level. Detection of this channel is described in [61].

It is important to distinguish between the access restrictions required by the security model and those supported by the hardware. In many security models, for example, a subject will have **read** access to objects at lower levels and **write** access to objects at higher levels. However, the conceptual objects of a security model may not correspond to the objects with which access rights are associated by the hardware. Also, the collection of permissible rights may not be the same. Here we are referring to the hardware support mechanisms. Implementing a secure system on top of these will almost always require a mapping from the required protection onto the supplied protection schemes.

Often, hardware enforced access rights are associated with pages or with segments. For instance, a scheme associating access rights with *physical* pages was the basis of protection in the IBM 360. However, it is generally desirable to map ‘‘objects’’ (in the sense of security models) and their associated access protection to virtual memory locations rather than to physical locations. The memory management system can map these in turn to physical locations in a way that is transparent to the programmer. The simplest way to accomplish this is to associate a single segment with each object. This permits the use of the hardware segment protection mechanisms directly by the security system. However, it requires that the hardware segmentation accommodate potentially a large number of segments of widely varying sizes. If, as is common, hardware-supported segments have a size limitation, there is a need to maintain multi-segment objects. This problem led to some difficulties in SCOMP [46] which supported only small segments.

3.3-E(4)-b Multiple Execution Modes.

Another common scheme is for each domain/process to operate at any time in some *execution mode*. Many operating systems, such as Unix, support two such modes: *system* and *user*. These are hierarchical; the *system* domain has all privileges of *user* domain and some additional privileges as well. This protection may be implemented very simply by having a mode bit in the process descriptor. In such systems, it is crucial to protect this bit from modification by the user process. The user in system mode has considerable privilege and essentially unlimited access to system resources.[16] Having only two modes is generally too coarse a level of protection for most security applications.

Implementing two execution domains is a special case of a *ring* architecture such as Multics [103]. Rings are a general hierarchical protection structure. A program executing in the innermost (lowest-numbered) ring has all system privileges. The higher the ring number, the fewer privileges it enjoys. Each ring is protected from those outside it. Thus, under the principle of ‘‘least privilege’’ a program should operate in the highest possible ring. Various machines have provided hardware support for rings (eg. the ICL 2900 and the later Honeywell Multics machines).

One issue with ring-based systems is: how many rings are enough? In a two-ring system, the granularity of protection may be too coarse to offer adequate isolation between processes and between parts. Systems have been proposed with up to 64 rings, though in practice fewer are used. The ICL 2900 machine supported sixteen rings, for example, most of which were used by the VME/B operating system. [47]

A ring architecture supports some types of security, but by itself does not necessarily support well the isolation of mutually suspicious domains, particularly if access rights are associated exclusively with the rings and not with the processes. Two processes either operate in the same ring or in different rings. If in the same ring, they have all the same permissions and may not be effectively isolated. If in different rings, the more privileged domain may be protected from the less privileged, but not vice versa. Ring-based systems also do not easily support the notion of a protected subsystem and other needs for non-hierarchical domains.

A more general mechanism is the association of access permissions with individual subject/object pairs. [76] This can be implemented by storing rights with the subject (capabilities) or with the object (access control lists), or in a general table indexed by subject/object pairs. Support for this basic concept

---

[16] The scheme for protecting system mode is an obvious candidate for formal verification. The formal verification of the KIT operating system [11] included a proof that no user-mode program could enter system mode. Popek and Farber [108] describe an implementation flaw in the PDP-10 Tenex that allowed a user to seize system mode by forcing a counter to overflow.

varies widely from the authorization lists associated with files in Unix to complete architectures based upon capabilities.

3.3-E(4)-c Capabilities.

Capability-based machines have often been cited as particularly well-suited to constructing secure systems [42, 82]. A capability is an unforgeable ‘‘ticket’’ that grants its holder a specific kind of access to an object. Capabilities, introduced by Dennis and VanHorn [36], are sometimes implemented as virtual addresses with additional bits that encode access rights. They are an efficient protection mechanism since they may be implemented via a natural extension to existing paged and segmented hardware architectures. On such machines, the current page and segmentation registers can be extended to hold access privileges. The concept has even been extended from virtual memory protection to file protection. [106]

A major problem is maintaining the integrity of the capabilities. They must be genuinely unforgeable. Schemes for enforcing this involve having a tagged architecture or separate designated segments for storing capabilities. Another problem is that capabilities allow control of access, but may not be well-matched for some types of security policies. [15] They do not support traceability of access since they do not answer the question ‘‘who has access to this object?’’[17] Also, passing of capabilities among domains may violate the containment goals of lattice-style security policies. Karger and Herbert [71] propose an augmented capability architecture designed to alleviate these difficulties. Capability based machines include the Plessy System 250, the Cambridge CAP Computer, IBM System/38, and Intel 432. Secure systems based on capabilities include the UCLA Secure Unix [107, 131], kernelized VM/370 [55], and PSOS [102].

3.3-E(4)-d Reference Monitor.

The most prevalent approach to implementing access control is the notion of a *reference monitor* [4]. A reference monitor is an access checking module that ideally:

- mediates all accesses by subjects to objects;
- can be verified to operate correctly; and,
- is tamperproof.

Some believe that ‘‘the reference monitor is inherent to the design of secure computers’’ [119]. Others question this dogma, arguing that there has been no convincing demonstration that a reference monitor is the only viable guarantor of security [22, 32, 40, 58]. Nevertheless, the reference monitor has been a fixture of all recent secure system implementations, and is likely to remain so since it is mandated by the evaluation criteria [99]. Reference monitors have been implemented both in software and in hardware. However, because of its pervasive role in the system, a hardware reference monitor is likely to be much more efficient.

An interesting implementation of domain separation mechanisms via a reference monitor is LOCK [119]. We discuss LOCK at some length here since it is one of the most recent and one of the best thought out approaches to implementing access control with the reference monitor concept. Moreover, LOCK is largely a hardware implementation of the reference monitor concept, which makes it particularly relevant to the current study.

The LOCK reference monitor is implemented as a co-processor called the SIDEARM—a separate embedded computer with isolated memory and processing power. The SIDEARM controls the resources of the host machine by mediating all access to those resources by users operating on the host CPU. The SIDEARM:

1. manages the identification and security labeling of all objects and subjects;
2. implements the mandatory security policy based on those identifications and security attributes; and,

[17] An early version of LOCK [20] used augmented capabilities that had both the object *and subject* recorded. This prevented passing of access rights between subjects.

3. is guaranteed not to be bypassed since it is physically impossible for the CPU to address its memory without going through the SIDEARM to obtain an object's address.

Moreover, all security information is stored on the SIDEARM, preventing tampering by user processes.

A novel aspect of the mandatory policy enforced by the SIDEARM is the LOCK *type-enforcement* policy. At any time, each subject is operating within a single ''domain.'' Associated with each object is a ''type.'' Domains are limited in access to certain types of objects; these limitations are statically defined by a system Domain Definition Table. Moreover, the ability of a subject to move from one domain to another is restricted according to a system Domain Transition Table, as described above in Section 3.3-D. The type-enforcement policy permits the construction of ''assured pipelines''—a mechanism of assuring that a data item passes through some series of transformations in a controlled manner without the possibility that the process may be subverted [18].[18] Type-enforcement is orthogonal to the lattice-based mandatory access policy and to the discretionary access policy both also enforced by the SIDEARM. An access is permitted only if it passes the ''filter'' of all three policies. Type enforcement is readily seen to be yet another use of access control to enforce domain separation.

3.4 Fault Detection and Handling

Unlike software, hardware can experience faults and failures relating to age, physical damage and external conditions of operation . Unless the possibility of faults is acknowledged in the design of the protection mechanisms, it may be possible for hardware faults to compromise the security of the system. Such a compromise may be exploited opportunistically by a malicious program, though such faults are unlikely to be predictable enough to form the basis of an attack.

A secure machine design should include facilities for fault detection and fault handling mechanisms. Such mechanisms include error detection and correction coding on memory, self-test modes for hardware, and protected restart mechanisms.

It is crucial that the design of error handling mechanism and particularly the use of existing hardware error features be considered carefully in secure system design. Since errors are considered exceptional conditions, formal models of the system may not include them; this is yet another instance of the need noted in Section 2.2 for complete models. Some standard responses to system errors—causing a complete memory dump—are inherently un-secure. Other mechanisms for error handling, particularly interrupts, are notoriously hard to model formally.

The design of security critical hardware can profit from the techniques of careful fault analysis as is used in the design of safety-critical systems [80]. These techniques aim at identifying potential faults that can lead to breaches of safety/security and providing backup mechanisms that will preserve system security even if they occur. This type of analysis can significantly reduce the likelihood that a single point failure can compromise system security. The National Security Agency's Security Fault Analysis (SFA), for example, requires that no single point of failure can cause a security violation.

Another approach to maintaining desired system behavior in the presence of potentially faulty hardware is to use fault-tolerant hardware design. Fault-tolerant design permits the correct operation of the system even in the presence of faulty hardware units. Fault-tolerance is often gained by the addition of redundant hardware units that vote on their results. Some schemes ensure the correct behavior of the system even in the presence of maliciously faulty units (Byzantine faults) [75]. Implementations of at least one such scheme have been modeled and formally verified to the hardware level. [12, 14] Despite an extensive literature on fault-tolerance, we are not aware of the use of fault tolerant hardware in secure system design, though the idea is an obvious one.

An alternative to redundancy is provided by an approach called ''dynamic verification''. This uses

---

[18] For example, such a ''pipeline'' can be used to transform raw-text to formatted-text to labeled-text to output-text. Type enforcement can ensure that a malicious user process cannot access labeled-text to change the security labels before it is output. Only the printer demon subject operates in a domain with access to labeled-text and then only **read** access. See [139] for an example of a verified labeler utility that uses type-enforcement to create and enforce such a pipeline.

independent hardware consistency checks each time specific critical actions are performed. The Berkeley PRIME system, for example, verifies pages access, allocation and clearing and disk cylinder access, allocation and clearing. [130] This approach is claimed to significantly reduce the amount of redundant hardware while achieving assurance that ''one user's information cannot become available to another user gratuitously even in the presence of a single hardware or software fault.'' [41]

3.5 I/O Access Control

I/O control is another area of potential concern for secure system developers. There are various reasons for this.

- Output devices are the last outpost of security. Once information has been printed/displayed/broadcast incorrectly it is effectively compromised.
- I/O devices are often shared across multiple security levels and given ''trusted'' status, i.e., permitted access not afforded to a typical user.
- Because i/o is traditionally a system performance bottleneck, designers have tended to make peripherals somewhat autonomous.

These factors make I/O routines and devices a likely candidate for attack. An example is noted by Rushby as follows.

> In its simplest form, the I/O security problem is that DMA devices generally have completely free access to the whole of memory. The elaborate steps taken to prevent one user program from accessing another's memory count for nothing if programs can call upon I/O devices to evade the protection mechanisms that restrict their own behavior. To avoid this type of security flaw, it is necessary either to exclude DMA devices altogether (which drastically reduces functionality and performance) or to require that all I/O requests are handled by trusted software (which greatly increases the quantity and complexity of trusted code in the system). [113]

DMA controllers are simply one instance of increasingly intelligent I/O devices. Many contain microprocessors and are capable of quite sophisticated behavior. How, for instance, do we guarantee that an autonomous smart printer demon does not change or delete the output security labels on a document?[19] Many attacks on system security using these ''smart'' devices can be thwarted by ensuring that memory accesses from I/O devices are subject to all of the same controls as other user ''subjects'', i.e., by ensuring that the I/O facilities are included as part of the complete system security model as advocated in Section 2.2 above. Another approach was suggested in Section 3.1 above; there we described how bulk encryption is used on LOCK to eliminate the need for much verification of I/O control by rendering the data non-sensitive. Research is needed into ways to ensure I/O security while preserving many of the benefits of DMA and other advances in I/O performance enhancement.

3.6 Processing Requirements

The performance of an information system is not a security concern, *per se*. However, if the protection mechanisms are so slow or cumbersome as to render the system unpleasant or inconvenient to use, this can have several bad effects.

- Users may attempt to circumvent the protection mechanisms and the quite significant contribution of users to system security will be reduced.
- System administrators may bow to the wishes of users and disable some or all security controls.
- The system may be viewed as commercially non-competitive with others that provide inferior security but enhanced performance.

Ideally, security mechanisms should be implemented in such a way that they have negligible performance

[19] The use of the LOCK type-enforcement mechanism has been suggested as one approach. [139]

impact and cause users minimal inconvenience. However, this is not always possible since some of the functions that users find convenient have potential security implications. Unix, for example, is often considered a hacker-friendly system because of the relative freedom of programmer access to system resources; a Unix super-user has essentially unlimited access. However, such freedom is antithetical to secure operation.

The use of hardware to implement protection mechanisms cannot alleviate the need to place some restrictions on access, but it can make the restrictions less onerous by significantly decreasing their impact on overall system performance. This must be considered in secure system development. LOCK's hardware based kernel approach is aimed at no more than 10% performance reduction over the unaltered hardware [119], compared to a 90% reduction for a software approach reported for KVM/370 [54].

Hardware that assists in speeding the processing of domains, checking security on accesses, or moving information securely within the system can enhance system performance and the usability of the system. Landwehr and Carroll [77] give a number of examples.

Some of the ways in which hardware designers have attempted to enhance system performance are troublesome from the point of view of verification and security. For example, instruction pipelines and cacheing schemes are aimed at enhancing performance by reducing the amount of time required for instruction execution. Most such schemes are logically transparent to the user program. However, some versions of the Motorola MC68000 have cacheing that actually alters the *logical* results of program execution with the result that it is very difficult to reason about programs on these machines.[20]

3.7 Physical Security

There are a variety of books such as [130] covering physical security of computing resources. Techniques that fall generally under physical security include:

- maintaining locks and alarm systems on computer rooms;
- maintaining (securely stored) backup copies of all media;
- having write inhibit mechanisms on disks and tapes and write-only media (such as laser disks) for audit data;
- assuring secure communications by protection of data paths and encryption;
- protecting against electronic eavesdropping by TEMPEST techniques.

These features are not generally candidates for formal analysis. However, there is no reason that a careful risk analysis could not be used to identify threats from the physical environment and lead to enhanced security.

4. Conclusions

Hardware has a tremendous impact upon the security of a system. Yet emerging techniques for enhancing the reliability of hardware have had little impact on secure system development. We feel that the security and hardware verification communities have a great deal to offer one another. We have suggested ways for determining appropriate security properties for hardware; these properties can serve as specification for hardware verification and provide an interesting and vital applications area for hardware verification work. The techniques of hardware verification can be a useful tool in enhancing system security and can be used in concert with the higher-level analysis techniques used to date.

We have also considered a number of hardware features that often directly impact secure system operation and are candidates for formal scrutiny. A continuing challenge will be finding ways to specify adequately

[20]Yuan Yu at the University of Texas is attempting to formalize the semantics of the MC68000 instruction set. He was forced to leave out some instructions because of the unpredictable effect of the cacheing mechanisms.

these types of hardware features and provide a firm formal foundation for reasoning about these specifications. Research in this area can and should ultimately contribute to the security of computing systems.

Acknowledgements

This work was sponsored in part at Computational Logic, Inc. by the Naval Research Laboratory, contract number N00014-90-C-2351. It benefited significantly from input from our contract monitors, particularly Andy Moore of NRL, and from review and comments by our colleagues at CLI, particularly Matt Kaufmann.

Appendix: Evaluating Hardware Support for System Security

Part of our initial goal for this report was to ‘‘list the vulnerabilities that hardware designs may exhibit and classify their relative importance.’’ However, we found this goal essentially impossible to accomplish. Hardware is an extremely flexible medium for the realization of secure system designs. With very few exceptions, the vulnerabilities of a hardware implemented secure system are the same as those of a software implemented secure system. The few exceptions have to do with the physical, rather than the logical properties of a secure system. Issues such as the need to protect against external interception of magnetic remanence simply do not arise at the logical level of concern.

We decided to present our list of vulnerabilities in the form of a collection of questions which might be asked of hardware implementations of secure systems. These questions define what we believe to be important concerns related to hardware in secure system development. The relative importance of these concerns is roughly as follows. The more widespread the reliance of system security on a hardware feature, the more important it is that that feature be implemented correctly. For example, it is important that the audit mechanisms of the system be implemented in a tamper-proof fashion. But it is more important that the memory management system maintain process isolation correctly, particularly if the isolation of ‘‘subjects’’ in the model depends directly upon the isolation of processes in the implementation.

General Concerns

- Are the security constraints justifiable and reasonable for the intended application of the system?
- Are the protection mechanisms well-structured and adequately designed to allow formal modeling and analysis?

System Architecture

- Is the structure of the hardware architecture conducive to proof?
- Does the structure of the architecture *match* the structure of the model? E.g., is it layered in such a way that the protection structures support the layering of security constraints mandated by the model?
- Is the security functionality encapsulated into a clearly delimited kernel?
- Is the system structured around a reference monitor? Is a reference monitor appropriate for the application?
- Does the structure of the architecture make clear the security perimeter? Are there features ostensibly outside the security perimeter which are potentially security-relevant?

Memory Management and Protection

- What memory protection is provided by the hardware?
- Do the security mechanisms rely upon the memory protection? Is it adequate to these purposes?
- Is the memory management entirely transparent from a user perspective? E.g., are the effects of page faults visible to the user? Could user activity force page faults in a predictable way?
- Are the hardware features which support memory management accessible to the user? E.g., are base and bounds registers adequately protected?
- Is the memory management scheme susceptible to formal modeling and proof? Is a clear axiomatization possible?
- If memory management is used to protect system software, is the protection strong enough for this purpose?
- Are the memory divisions supported by the machine of adequate granularity and/or size to support ‘‘objects’’ in the model?

Execution Domains

- What are the execution domains provided by the machine? How do they map onto the domains mandated by the security model? Are they adequate to support the model?
- Does the hardware support multiple execution domains in an efficient manner?

Processes

- What is the appropriate mapping between hardware-supported processes and the execution domains of the model?
- Does the hardware support efficient process swapping? How much information is stored upon a process swap? Is the information for all suspended processes adequately protected from manipulation by the currently running process?

Initialization

- What are the mechanisms for establishing a consistent initial state of the machine? Is the initial state secure? Is the initial state adequate for establishing other domains securely?
- Does the initial state require ‘‘trusted’’ capabilities?
- Does the hardware support protection of the initialization facilities? Are the bootstrap routines and security tables adequately protected when the system is not running?
- Would separate initialization hardware be appropriate for the application?
- Is the initialization problem considered in any formal model of the system?

Linking Users with Domains

- Are the actions of each execution domain traceable to a specific responsible user?
- Is there hardware support for linking users with domains? E.g., is there some hardware mechanism supporting authentication of users at login time?
- Is each user periodically reauthenticated?
- Is adequate audit data maintained to assure that security violations can be scrutinized and flaws corrected?
- Does the hardware support the collection of audit data to minimize its operational impact?
- Is the audit data adequately protected from tampering? Is the audit data protected differently from other data in the system?

Communication Between Domains

- How do domains communicate? What is the hardware support for this?
- Is there hardware support for restricting domain changes?
- Is there support for calling a more restricted domain (kernel call)? If so, does it support address validation and parameter checking?
- If a message passing system, how is message passing implemented and restricted?
- Is the communication mechanism amenable to modeling formally?

Isolation of Domains

- Which mechanism is employed for isolation of domains: temporal, physical, cryptographic, or logical?
- If temporal isolation is employed, is there adequate protection against residual information remaining between uses? E.g., is the memory scrubbed between uses? Is there hardware support for this?

- Are boot-strap routines, initialization software, and other system software adequately protected to prevent sensitive information being preserved from one use to the next?
- Are removable storage media sanitized between uses?
- If physical isolation is employed, are the remaining channels completely enumerated in the model? E.g., are all bypasses accounted for and monitored?
- Does physical isolation really make system penetration more difficult or less profitable?
- Schemes employing physical isolation sometimes require special hardware such as physical one-way links? Are these devices adequate to the purposes?
- If cryptographic isolation, is the coding stream strong enough to provide adequate protection?
- Is the cryptographic hardware correctly implemented and correctly integrated into the larger system? E.g., can any cleartext bypass be exploited to compromise system security? Is sensitive data always encrypted as required?
- How is key data protected in the system? How is it distributed?
- Is end-to-end encryption used to protect data transmission?
- If cryptographic techniques are used to close covert channels, are they logically strong enough to do so? E.g., if the channel only depends upon the *dissimilarity* of two values, encrypting them may not mask this.
- If logical isolation is used, how is it implemented?
- Is there hardware support for storing and checking access permissions? Do the hardware supported permissions match the access permissions mandated by the model?
- Does the hardware and operating system support multiple execution modes? Are these helpful in supporting the protection structures of the model? Are there enough of them? Are they appropriately hierarchical?
- Are there adequate protections to keep a user from gaining inappropriate permissions? Is this provable?
- How are access permissions stored? Are they associated with the subject or with the object?
- If the underlying hardware support is a capability system, how does the protection mechanism overcome inherent limitations of capabilities? How is the integrity of the capabilities maintained?
- If a reference monitor is used, does it satisfy the conditions of mediating all access, being verifiable, and tamper-proof?
- If the monitor is not implemented in hardware, does it provide adequate performance?
- Is there any mechanism for constructing ‘‘assured pipelines’’ in the system?

Fault Detection and Handling

- Can a single hardware fault cause a security compromise?
- Does the hardware provide support for fault detection and recovery?
- Have the error detection and recovery mechanisms been considered in the system design? Can they be modeled and included in the formal model of the system?
- Are interrupts used for error handling? If so, have their effects on security-relevant computing been investigated?
- Has the system design been subjected to a careful fault analysis?
- Has fault-tolerant hardware been used in the system implementation? If so, has any formal analysis been used to show that the system is appropriately ‘‘tolerant’’ of faults?

I/O Access Control

- Do the I/O devices require exceptional ‘‘trust’’ in the system design?
- Is each I/O device’s access to memory mediated by the security mechanisms of the system? Is DMA permitted in a fashion outside the standard security mechanisms?
- Is the behavior of smart I/O controllers adequately controlled?
- Is classified output appropriately protected from modification by smart I/O controllers?

Processing Requirements

- Do the security mechanisms so degrade system performance that users will try to circumvent them? Do they render the system commercially non-competitive?
- Is hardware used effectively to ameliorate the performance impact of security restrictions?
- Do attempts to gain enhanced performance alter the logical behavior of the machine in any way which makes it very difficult to model?

Physical Security

- Is physical security adequate?
- Are there adequate locks and alarm systems on the secure facilities?
- Are there securely stored backup copies of all media?
- Are there appropriate write inhibit mechanisms on disks and tapes and write-only media (such as laser disks) for audit data?
- Is encryption and physical security on data paths used to assure secure communications?
- Are there safeguards (e.g., TEMPEST techniques) against electronic eavesdropping?

References

[1] R.P. Abbott, *et. al.*.
Security Analysis and Enhancements of Computer Operating Systems.
Technical Report S-413558-740, National Bureau of Standards,
1974.

[2] Air Force Studies Board.
Committee on Multilevel Data Management Security.
Washington, D.C., 1983.

[3] S.R. Ames and D.R. Ostreicher.
Design of a Message Processing System for a Multilevel Secure Environment.
In *Proceedings of the National Computer Conference, Vol. 47*, Pages 765-771. AFIPS, 1978.

[4] J.P. Anderson.
Computer Security Technology Planning Study.
Technical Report ESD-TR-73-51, Volumes I and II, Electronic Systems Division, Bedford, MA.,
October, 1972.

[5] J.C. Barros and B.W. Johnson.
Equivalence of the Arbiter, the Synchronizer, the Latch, and the Inertial Delay.
IEEE Transactions on Computers:603-614, July, 1983.

[6] D.E. Bell.
Concerning ‘Modeling’ of Computer Security.
In *Symposium on Security and Privacy*, Pages 8-13. IEEE, 1988.

[7] D.E. Bell and L.J. LaPadula.
Secure Computer System: Unified Exposition and Multics Interpretation.
Technical Report MTR-2997, MITRE Corp., Bedford, Mass., July, 1975.

[8] H.K. Berg, W.E. Boebert, W.R. Franta, T.G. Moher.
Formal Methods of Program Verification and Specification.
Prentice-Hall, Englewood Cliffs, N.J., 1982.

[9] W.R. Bevier, W.A. Hunt, Jr., J S. Moore, W.D. Young.
An Approach to Systems Verification.
*Journal of Automated Reasoning*5(4):411-428, December, 1989.

[10] W.R. Bevier.
Kit and the Short Stack.
*Journal of Automated Reasoning*5(4):519-530, December, 1989.

[11] W.R. Bevier.
Kit: A Study in Operating System Verification.
*IEEE Transactions on Software Engineering*15(11):1368-81, November, 1989.
Also published as CLI Technical Report 28.

[12] W.R. Bevier, W.D. Young.
The Proof of Correctness of a Fault-Tolerant Circuit Design.
In *Proceedings of the Second International Working Conference on Dependable Computing for Critical Applications*, Pages 107-114. IFIP, February, 1991.

[13] K.J. Biba.
Integrity Considerations for Secure Computer Systems.
Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA.,
April, 1977.

[14] M. Bickford and M. Srivas.
Formal Verification of a Fault-Tolerant Microprocessor System Design.
NASA Contractor Report , Odyssey Research Associates,
February, 1991.

[15] W.E. Boebert.
On the Inability of an Unmodified Capability Machine to Enforce the *-Property.
In *Proceedings of the Seventh National Computer Security Conference*, September, 1984.

[16] W.E. Boebert.
The LOCK Demostration.
In *Proceedings of the 11th National Computer Security Conference*, National Institute of Standards and Technology, 1988.

[17] W.E. Boebert.
Constructing and Infosec System using LOCK Technology.
distributed at the LOCK Tutorial, 11th National Computer Security Conference.

[18] W.E. Boebert and R.Y. Kain.
A Practical Alternative to Hierarchical Integrity Policies.
In *Proceedings of the 8th National Computer Security Conference*, IEEE, October, 1985.

[19] W.E. Boebert, R.Y. Kain, W.D. Young.
Secure Computing: The Secure Ada Target Approach.
*Scientific Honeyweller*6(2):1-17, July, 1985.

[20] W.E. Boebert, W.D. Young, R.Y. Kain, S.A. Hansohn.
Secure ADA Target: Issues, System Design, and Verification.
In *Proceedings of the Symposium on Security and Privacy*, IEEE, 1985.

[21] D. Bonyun.
A New Model of Computer Security with Integrity and Aggregation Considerations.
Technical Report , I.P. Sharpe,
March, 1978.

[22] D. Bonyun.
The Use of Architectural Principles in the Design of Certifiably Secure Systems.
*Computers and Security*2:153-162, 1983.

[23] R. S. Boyer and J S. Moore.
Proof Checking the RSA Public Key Encryption Algorithm.
*American Mathematical Monthly*91(3):181-189, 1984.

[24] B.C. Brock and W.A. Hunt, Jr.
Report on the Formal Specification and Partial Verification of the VIPER Microprocessor.
Technical Report 46, Computational Logic, Inc.,
June, 1989.

[25] R.E. Bryant.
A Switch-Level Model and Simulator for MOS Digital Systems.
IEEE Transactions on Computers(2):160-177, February, 1984.

[26] W.E. Burr, A.H. Coleman, and W.R. Smith.
Overview of Military Computer Family Architecture Selection.
In *National Computer Conference Proceedings, Volume 46*, Pages 131-137. AFIPS, 1977.

[27] J.J. Carnall and A.F. Wright.
Secure Communications Processor—Hardware Verification Report.
Technical Report ARPA Order Number 3373, Program Code No. 7P10, Honeywell Information Systems, Inc., Federal Systems Division,
1979.

[28] K.M. Chandy and J. Misra.
Parallel Program Design, A Foundation.
Addison Wesley, 1988.

[29] L.J.M. Claesen, editor.
Formal VLSI Specification and Synthesis.
North-Holland, Amsterdam, 1990.

[30] D.D. Clark and D.R. Wilson.
A Comparison of Commercial and Military Computer Security Policies.
In *Proceedings of the Symposium on Security and Privacy*, Pages 184-194. IEEE, 1986.

[31] A. Cohn.
The Notion of Proof in Hardware Verification.
*Journal of Automated Reasoning*5(2):127-139, June, 1989.

[32] G.I. Davida, R.A. DeMillo, R.J. Lipton.
A System Architecture to Support a Verifiably Secure Multilevel Security System.
In *Proceedings of the Symposium on Security and Privacy*, Pages 137-144. IEEE, 1984.

[33] R.A. DeMillo, J. Lipton, and A.J. Perlis.
Social Processes and Proofs of Theorems and Programs.
*Comm. ACM*22(5):271-280, 1979.

[34] D. Denning.
A Lattice Model of Secure Information Flow.
*CACM*19(5):417-429, 1976.

[35] D. Denning.
Cryptography and Data Security.
Addison-Wesley, Reading, MA, 1982.

[36] J.B. Dennis and E.C. VanHorn.
Programming Semantics for Multiprogrammed Computations.
*Communications of the ACM*9(3):143-155, March, 1966.

[37] Y. Deswarte, J.-C. Fabre, J.-C. Laprie, and D. Powell.
A Saturation Network to Tolerate Faults and Intrusions.
In *Proceedings of the 5th Symposium in Distributed Software and Database Systems*, Pages 74-81. IEEE, January, 1986.

[38] E.W. Dijkstra.
The Structure of the 'THE' Multiprogramming System.
*Communications of the ACM*11(5), May, 1968.

[39] B.B. Dillaway and J.T. Haigh.
A Practical Design for a Multilevel Secure Database Management System.
In *Proceedings of the 2nd Aerospace Computer Security Conference*, AIAA, 1986.

[40] J.E. Dobson and B. Randell.
Building Reliable Secure Computing System out of Unreliable Insecure Components.
In *Proceedings of the Symposium on Security and Privacy*, Pages 187-193. IEEE, 1986.

[41] R.S. Fabry.
Dynamic Verification of Operating System Decisions.
*Communications of the ACM*16(11), November, 1973.

[42] R.S. Fabry.
Capability-based Addressing.
*CACM*17, 7, July, 1974.

[43] H. Feistel, W.A. Notz, J.A. Smith.
Some Cryptographic Techniques for Machine to Machine Data Communications.
*Proceedings IEEE*63(11):1545-1554, 1975.

[44] C.T. Ferguson and C.B. Murphy.
A Proposed Policy for Dynamic Security Lattice Management.
In *Proceedings of the 9th National Computer Security Conference*, 1986.

[45] J.H. Fetzer.
Program Verification: The Very idea.
*Communications of the ACM*31(9):1048-1063, September, 1988.

[46] L.J. Fraim.
SCOMP: A Solution to the MLS Prolem.
*IEEE Computer*16(7), July, 1983.

[47] M. Gasser.
Building a Secure Computer System.
Van Nostrand Reinhold, New York, 1988.

[48] V. Gligor.
The Verification of the Protection Mechanisms of High-Level Language Machines.
*International Journal of Computer and Information Sciences*12(4):211-246, 1983.

[49] V. Gligor.
Hardware Selection and Evaluation Guidelines for Trusted Computer Systems.
Draft , Dept. of Electrical Engineerings, University of Maryland,
November, 1984.

[50] V. Gligor.
Analysis of the Hardware Verification of the Honeywell SCOMP.
In *Proceedings of the Symposium on Security and Privacy*, Pages 32-41. IEEE, 1985.

[51] V. Gligor.
On Denial-of-Service in Computer Networks.
In *Proceedings of the International Conference on Data Engineering*, Pages 608-617. February, 1986.

[52] J.A. Goguen and J. Meseguer.
Security Policy and Security Models.
In *Proceedings of the Symposium on Security and Privacy*, Pages 11-20. IEEE, 1982.

[53] J.A. Goguen and J. Meseguer.
Unwinding and Inference Control.
In *Proceedings of the Symposium on Security and Privacy*, Pages 75-86. IEEE, 1984.

[54] B.D. Gold, *et. al.*.
KVM/370 in Retrospect.
In *Proceedings of the Symposium on Security and Privacy*, IEEE, 1984.

[55] B.D. Gold, *et. al*.
A Security Retrofit of VM/370.
In *Proceedings of the National Computer Conference, Vol. 48*, Pages 417-429. AFIPS, 1979.

[56] R.P. Goldberg.
Survey of Virtual Machine Research.
*IEEE Computer*7(6):33-45, June, 1974.

[57] D.I. Good.
Structuring a System for A1 Certification.
Internal Note 145, Institute for Computing Science, The University of Texas at Austin,
August, 1984.

[58] D.I. Good.
The Foundations of Computer Security: We Need Some.
Internal Note, Computational Logic, Inc., September, 1986.

[59] D.I. Good and W.D. Young.
Mathematical Methods for Digital Systems Development.
Technical Report technical report 67, Computation Logic, Inc.,
August, 1991.

[60] S. Graham and P. Denning.
Protection—Principles and Practice.
In *Proceedings AFIPS SJCC, Vol. 40*, Pages 417-429. ADIPS Press, 1972.

[61] J.T. Haigh, J. McHugh, R. Kemmerer, W.D. Young.
An Experience Using Two Covert Channel Analysis Techniques on a Real System Design.
In *Symposium on Security and Privacy*, IEEE, April, 1986.

[62] B. Hebbard, *et. al.*.
A Penetration Analysis of the Michigan Terminal System.
*ACM Operating Systems Review*14(1):7-20, 1980.

[63] J.L. Hennessy and D.A. Patterson.
Computer Architecture: A Quantitative Approach.
Morgan Kaufmann, San Mateo, CA., 1990.

[64] C.G. Hoch.
Hardware Support for Modern Software Concepts.
PhD thesis, The University of Texas at Austin, 1978.

[65] L.J. Hoffman.
The Formulary Model for Access Control and Privacy in Computer Systems.
Technical Report SLAC-117, UC-32, Stanford University,
May, 1970.

[66] W. A. Hunt, Jr.
The Mechanical Verification of a Microprocessor Design.
In D. Borrione (editor), *From HDL Descriptions to Guaranteed Correct Circuit Designs*, pages 89-132. North Holland, 1987.

[67] Hunt, W.A., Jr., and W.D. Young.
Maintaining Abstractions with Verification.
In *Proceedings of the Fifth Annual Conference on Computer Assurance*, IEEE COMPASS ’90, Gaithersburg, Maryland, June, 1990.

[68] F. Jahanian and A.K. Mok.
Safety Analysis of Timing Properties in Real-Time Systems.
*IEEE Transactions on Software Engineering*12(9), September, 1986.

[69] J. Joyce, G. Birtwistle, and M. Gordon.
Proving a Computer Correct in Higher Order Logic.
Technical Report , University of Calgary, Department of Computer Science,
August, 1985.

[70] Jeffrey J. Joyce.
Generic Specification of Digital Hardware.
Technical Report 90-27, The University of British Columbia, Department of Computer Science,
September, 1990.

[71] P.A. Karger and J.A. Herbert.
An Augmented Capability Architecture to Support Lattice Security and Traceability of Access.
In *Proceedings of the Symposium on Security and Privacy*, Pages 2-12. IEEE, 1984.

[72] M. Kaufmann.
A Hardware Reset Lemma and its Proof.
Internal Note 230, Computational Logic, Inc.,
May, 1991.

[73] J. Keeton-Williams, S.R. Ames, B.A. Hartman, and R.C. Tyler.
Verification of the ACCAT-Guard Downgrade Trusted Process.
Technical Report NTR-8463, The Mitre Corporation, Bedford, MA., 1982.

[74] T. Kibalo and W.E. Boebert.
Using Embedded COMSEC: An Integrator’s Viewpoint.
In *Proceedings of the 1st Mid-Atlantic Intelligence Symposium*, AFCEA, 1986.

[75] L. Lamport, R. Shostak, and M. Pease.
The Byzantine Generals Problem.
*ACM TOPLAS*4(3):382-401, July, 1982.

[76] B.W. Lampson.
Protection.
In *Proceedings 5th Princeton Symposium on Info. Sci. and Syst.*, Princeton University, March, 1971.

[77] C.E. Landwehr and J.M. Carroll.
Hardware Requirements for Secure Computer Systems: A Framework.
In *Symposium on Security and Privacy*, IEEE, 1984.

[78] C.E. Landwehr, B. Tretick, J.M. Carroll, and P. Anderson.
A Framework for Evaluating Computer Architectures to Support System with Security Requirements, with Applications.
Technical Report NRL Report 9088, Naval Research Laboratory,
November, 1987.

[79] M. Leeser and G. Brown, editors.
Hardware Specification, Verification and Synthesis: Mathematical Aspects.
Springer-Verlag, Lecture Notes in Computer Science 408, Berlin, 1989.

[80] N.G. Leveson.
Software Safety: Why, What, and How.
In *ACM Computing Surveys*, Pages 125-163. June, 1986.

[81] K. Levitt, S. Crocker, and D. Craigen.
Introduction to VERKshop III.
*ACM Software Engineering Notes*10(4):ii-iii, February, 1985.

[82] H. Levy.
Capability-Based Computer Systems.
Digital Press, Bedford, MA, 1983.

[83] R.R. Linde.
Operating System Penetration.
In *National Computer Conference, Vol. 44*, AFIPS, 1975.

[84] L.G. Marcus and B.H. Levy.
Specifying and Proving Core VHDL Descriptions in the State Delta Verification System.
Technical Report ATR-89(4778)-5, Aerospace Corporation,
September, 1989.

[85] E.J. McCauley and P.J. Drongowski.
KSOS: The Design of a Secure Operating System.
In *Proceedings of the National Computer Conference, Vol. 48*, Pages 345-353. AFIPS, 1979.

[86] D. McCullough.
Foundations of Ulysses: The Theory of Security.
Technical Report RADC-TR-87-222, Rome Air Development Center,
July, 1988.

[87] D. McCullough.
A Hookup Theorem for Multilevel Security.
*IEEE Transactions on Software Engineering*16(6):563-568, 1990.

[88] J. McLean.
A Formal Statement of the MMS Security Model.
In *Proceedings of the Symposium on Security and Privacy*, Pages 188-194. IEEE, April, 1984.

[89] J. McLean.
Reasoning About Security Models.
In *Symposium on Security and Privacy*, Pages 123-131. IEEE, 1987.

[90] T. Melham.
Abstraction Mechanisms for Hardware Verification.
In G. Birtwistle and P.A. Subrahmanyam (editor), *VLSI Specification, Verification, and Synthesis,* pages 129-157. Kluwer Academic Publishers, Boston, 1988.
reprinted in Yoeli90.

[91] Ministry of Defence.
Interim Defence Standard 00-55.
D/D Stan/303/12/3.

[92] R.R. Moeller.
Computer Audit, Control, and Security.
J. Wiley & Sons, Somerset, N.J., 1989.

[93] L.M. Molho.
Hardware Aspects of Secure Computing.
In L.J. Hoffman (editor), *Security and Privacy in Computer Systems*. Melville Publishing Co., Los Angeles, CA, 1973.

[94] A.P. Moore.
Using CSP to Develop Trustworthy Hardware.
In *Compass '90*, Pages 126-134. June, 1990.

[95] A.P. Moore.
The Specification and Verified Decomposition of System Requirements Using CSP.
*IEEE Transactions on Software Engineering*16(9), September, 1990.

[96] R. Morris and K. Thompson.
Password Security: A Case Study.
*CACM*22(11):594-597, 1979.

[97] National Research Council.
Computers at Risk.
National Academy Press, Washington, D.C., 1991.

[98] National Security Agency.
Information Systems Security Products and Services Catalogue.
Issued quarterly, January, 1989 and successors.
This is the endorsed tools list of the NCSC, which contains GVE and FDM.

[99] National Computer Security Center.
Department of Defense Trusted Computer System Evaluation Criteria.
Department of Defense Computer Security Center. 1985.
DoD 5200.28-STD.

[100] National Computer Security Center.
Department of Defense Trusted Network Interpretation.
Department of Defense Computer Security Center. 1987.
NCSC-TG-005 Version 1.

[101] R.B. Neely and J. Freeman.
Structuring Systems for Formal Verification.
In *Proceedings of the Symposium on Security and Privacy*, Pages 2-13. IEEE, 1985.

[102] P.G. Neumann, *et. al*.
A Provably Secure Operating System: The System, Its Applications, and Proofs.
Technical Report , SRI International,
February, 1977.

[103] E.I. Organick.
The Multics System: An Examination of Its Structure.
MIT Press, Cambridge, Mass., 1972.

[104] E.I. Organick.
Computer System Organization: The B5700/B6700 Series.
Academic Press, New York, 1973.

[105] J. Picciotto.
The Design of an Effective Auditing Subsystem.
In *Proceedings of the Symposium on Security and Privacy*, Pages 13-22. IEEE, 1987.

[106] G.J. Popek.
Protection Structures.
*IEEE Computer*7(6):22-33, June, 1974.

[107] G.J. Popek, et. al.
UCLA Secure Unix.
In *Proceedings of the National Computer Conference, Vol. 48*, Pages 355-364. AFIPS, 1979.

[108] G.J. Popek and D.A. Farber.
A Model for Verification of Data Security in Operating Systems.
*CACM*21(9), 1978.

[109] G.J. Popek and R.P. Goldberg.
Formal Requirements for Virtualizable Third Generation Architectures.
*Communication of the ACM*17(7):412-421, July, 1974.

[110] G.J. Popek and C.S. Kline.
Verifiable Secure Operating System Software.
In *Proceedings of the National Computer Conference, Vol. 43*, Pages 1450151. AFIPS, 1974.

[111] Romulus Staff.
Romulus Final Report--Draft Version.
Technical Report , Odyssey Research Associates,
1988.

[112] J. Rushby.
The Design and Verification of Secure Systems.
In *Proceedings of the 8th ACM Symposium on Operating System Principles*, Pages 12-20. ACM, December, 1981.

[113] J. Rushby.
Certifiably Secure Systems: Technologies, Prospects and Proposals.
Report prepared for the United States Office of Scientific Research, European Office of Aerospace Research and Development, August 8, 1982.

[114] J. Rushby.
Computer Security Models.
Draft Report, April 17, 1984, SRI International, Computer Science Laboratory.

[115] J. Rushby.
Foundations for Computer Security.
presented at the ACM SIGSAC meeting on "Requirements for Foundations for Computer Security", UCLA, November, 1987.

[116] J. Rushby and B. Randell.
A Distributed Secure System.
*IEEE Computer*16(7):55-68, 1983.

[117] A. Salem and D. Borrione.
Automatic Formal Verification of VHDL Descriptions: a First Prototype.
Research Report 823-I, Laboratoire Artemis, Grenoble, France,
June, 1990.

[118] J.H. Saltzer.
Protection and the Control of Information Sharing in Multics.
*Communications of the ACM*17(7):388-402, July, 1974.

[119] O.S. Saydjari, J.M. Beckman, J.R. Leaman.
LOCKing Computers Securely.
In *Proceedings of the 10th National Computer Security Conference*, Baltimore, MD, 1987.

[120] M.D. Schroeder, D.D. Clark, and J.H. Saltzer.
The Multics Kernel Design Project.
In *Proceedings of the 6th Symposium on Operating System Principles*, Pages 43-56. ACM, 1977.

[121] A.E. Siebert, D.I. Good.
General Message Flow Modulator.
Technical Report 42, Institute for Computing Science, The University of Texas at Austin, March, 1984.

[122] L. Smith.
Architectures for Secure Computing Systems.
Technical Report MTR-2772, Mitre Corp., Bedford, MA., April, 1975.

[123] M.K. Smith, D.I. Good, B.L. DiVito.
Using the Gypsy Methodology
Computational Logic Inc., 1986.
Revised January, 1988.

[124] M. K. Smith, A. Siebert, B. DiVito, and D. Good.
A verified encrypted packet interface.
*Software Engineering Notes*6(3), July, 1981.

[125] M.J. Spier.
A Model Implementation for Protective Domains.
*International Journal of Computer and Information Sciences*2(3), 1973.

[126] M.J. Spier, T.N. Hastings, and D.N. Cutler.
An Experimental Implementation of the Kernel/Domain Architecture.
*Operating System Review*7(4), 1973.

[127] J. Staunstrup, editor.
Formal Methods for VLSI Design.
North-Holland, Amsterdam, 1990.

[128] D. Sutherland.
A Model of Information.
In *Proceedings of the 9th National Computer Security Conference*, National Bureau of Standards, 1986.

[129] J.D. Tangney.
Minicomputer Architectures for Effective Security Kernel Implementations.
Technical Report ESD-TR-78-170, Mitre Corporation, Bedford, MA, October, 1978.

[130] B.J. Walker and I.F. Blake.
Computer Security and Protection Structures.
Dowden, Hutchinson & Ross, Stroudsburg, PA, 1977.

[131] B.J. Walker, R.A. Kemmerer, G.J. Popek.
Specification and Verification of the UCLA Unix Security Kernel.
*CACM*23(2), 1980.

[132] K.G. Walter, *et.al*.
Primitive Models for Computer Security.
Technical Report ESD-TR-74-117, Electronic Systems Division, Hanscom AFB, January, 1974.

[133] D. Weise.
Functional Verification of MOS Circuits.
In *Proc. 24th Design Automation Conference*, Pages 265-270. IEEE Computer Society Press, 1987.

[134] C. Weissman.
Security Controls in the ADEPT-50 Time-Sharing System.
In *Fall Joint Computer Conference Proceedings*, Pages 119-133. AFIPS, 1969.

[135] A.L. Wilkinson, *et. al.*.
A Penetration Study of a Burroughs Large System.
*ACM Operating Systems Review*15(1):14-25, 1981.

[136] M. Yoeli, editor.
Formal Verification of Hardware Design.
IEEE Computer Society Press Tutorial, 1990.

[137] Young, W.D, W.E. Boebert, R.Y. Kain.
Proving a Computer System Secure.
*Scientific Honeyweller*6(2):18-27, July, 1985.

[138] W.D. Young, J. McHugh.
Coding for a Believable Specification to Implementation Mapping.
In *Proceedings of the 1987 Symposium on Security and Privacy*, IEEE, 1987.

[139] W.D. Young, P.A. Telega, W.E. Boebert, R.Y. Kain.
A Verified Labeller for the Secure Ada Target.
In *Proceedings of the 9th National Computer Security Conference*, Gaithersburg, MD., September, 1986.

Table of Contents

1. Introduction 1
2. Defining Security for Hardware 2
 2.1. Security Models 2
  2.1-A. Types of Security Models 3
  2.1-B. Properties of Security Models 4
 2.2. Security Properties of System Components 5
  2.2-A. Determining the Security Perimeter 5
  2.2-B. Coverage of the Top-Level Model 6
  2.2-C. Modeling from the Top Down 8
 2.3. Formal Verification and Security 9
 2.4. The Role of Hardware Verification 10
3. Aspects of Hardware Support for Secure Systems 12
 3.1. Influence of System Architecture 13
 3.2. Memory Protection 14
 3.3. Execution Domains 14
  3.3-A. Processes 15
  3.3-B. Initialization 15
  3.3-C. Linking Users with Domains 16
  3.3-D. Communication Between Domains 17
  3.3-E. Isolation of Domains 17
   3.3-E(1). Temporal Isolation of Domains 18
   3.3-E(2). Physical Isolation 18
   3.3-E(3). Cryptographic Isolation 19
   3.3-E(4). Logical Isolation 19
    3.3-E(4)-a. Isolation via Access Permissions 19
    3.3-E(4)-b. Multiple Execution Modes 20
    3.3-E(4)-c. Capabilities 21
    3.3-E(4)-d. Reference Monitor 21
 3.4. Fault Detection and Handling 22
 3.5. I/O Access Control 23
 3.6. Processing Requirements 23
 3.7. Physical Security 24
4. Conclusions 24

Appendix: Evaluating Hardware Support for System Security 26

List of Figures

Figure 1:	Implementation of Abstractions	8
Figure 2:	Current Secure System Verification	10
Figure 3:	Proposed Secure System Verification	11